THE COMPUTER CONTROL OF CHANGING PICTURES

Gregory F. Pfister

Massachusetts Institute of Technology

Prepared for:

Office of Naval Research
Advanced Research Projects Agency

September 1974

BIBLIOGRAPHIC DATA SHEET	1. Report No. MAC TR- 135	2.	3. Recipient's Accession No. AD 787795
4. Title and Subtitle: The Computer Control of Changing Pictures			5. Report Date: Issued September 1974 6.
7. Author(s): Gregory F. Pfister			8. Performing Organization Rept. No. MAC TR- 135
9. Performing Organization Name and Address: PROJECT MAC; MASSACHUSETTS INSTITUTE OF TECHNOLOGY: 545 Technology Square, Cambridge, Massachusetts 02139			10. Project Task Work Unit No. 11. Contract Grant No. N00014-70-A-0362-0006
12. Sponsoring Organization Name and Address: Office of Naval Research, Department of the Navy, Information Systems Program, Arlington, Va 22217			13. Type of Report & Period Covered: Interim Scientific Report 14.

15. Supplementary Notes

Ph.D. Thesis, M.I.T. Department of Electrical Engineering, August 9, 1974

16. Abstracts

This document describes DALI (Display Algorithm Language Interpreter), a special-purpose programming language for the creation and control of changing pictures which exhibit complex static and dynamic interactions among their elements. DALI allows complex organizations if interpolated ("smooth") change, discrete change, and change in the structure of a picture to be generated in a modular way, in the sense that picture elements determine their own behavior and hence manner of change.

17. Key Words and Document Analysis. 17a. Descriptors

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Approved for Public Release; Distribution Unlimited	UNCLASSIFIED	264
	20. Security Class (This Page) UNCLASSIFIED	22. Price 8.50

THE COMPUTER CONTROL OF CHANGING PICTURES

Gregory F. Pfister

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2095 which was monitored by ONR Contract No. N00014-70-A-0362-0006

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

The Computer Control of Changing Pictures

by

Gregory F. Pfister

Submitted to the Department of Electrical Engineering
on August 9, 1974 in partial fulfillment of
the requirements for the Degree of Doctor of Philosophy

ABSTRACT

This document describes DALI (Display Algorithm Language Interpreter), a special-purpose programming language for the creation and control of changing pictures which exhibit complex static and dynamic interactions among their elements. DALI allows complex organizations of interpolated ("smooth") change, discrete change, and change in the structure of a picture to be generated in a modular way, in the sense that picture elements determine their own behavior and hence manner of change.

In DALI, pictures are composed of elements called picture modules. These are analogous to procedural activations or processes, and contain arbitrary event-driven procedures called daemons. Daemons are run under the control of global scheduling rules based on the functional dependence of daemons on one another. These rules result in smooth inter-daemon (process) communication and cooperation with no implicit or explicit reference to semaphores or other synchronization primitives in user code, while at the same time providing for a high degree of parallelism. Circular inter-daemon functional dependence results in iteration or relaxation. The environment structure used is predominantly stack-oriented.

THESIS SUPERVISOR: Michael L. Dertouzos

TITLE: Professor of Electrical Engineering

Acknowledgments

As is the case with any task the size of a doctoral dissertation, there are many persons who provided help and encouragement during the course of this work.

First and foremost among these is Professor Michael Dertouzos, my thesis advisor and chairman of my thesis committee. Without his continuing interest both in my research topic and in me personally, it is certain that the reported work would not have been as successful as it was. Professor Dertouzos was particularly helpful in the task of re-organizing my snarl of ideas for comprehensible exposition, a process which considerably increased my own comprehension of the issues involved and thus led to a better treatment of many topics.

A debt of thanks is also due to Professor J. C. R. Licklider, a member of my thesis committee, for his patience and support of my presence in his research group through a two-year fallow period during which I did not really know what I was doing, where I was going, or how I might manage to get there. This period was quite valuable in that it provided an opportunity for turning a fair amount of "book learning" into usable skills, attitudes, and ways of looking at problems. Professor Licklider also provided the computer facilities for an experimental implementation of DALI and for the seemingly infinite amount of editing this dissertation required.

Professor Carl Hewitt, the third member of my thesis committee, must be thanked for a large number of rambling conversations on topics as vague and general as "the nature of computation", conversations which helped crystallize several of my more fuzzy ideas to the point where they could be effectively used in this work.

Particular thanks must also be given to D. Austin Henderson, Jr., a

fellow graduate student, fellow computer graphicist, and fellow charter member of The Oldest Permanently Established Floating Bull Session In Project MAC. Computer graphics was not, shall we say, the hottest topic going in Project MAC while I was there; and so I am doubly grateful to Austin for providing technical companionship in his understanding of the computer graphical and linguistic issues I was trying to face, especially during the many periods of difficulty and discouragement that quite naturally occured in the course of this work.

Many other denizens of Project MAC are also to be thanked, both for technical aid and for simple companionship. Clearly included among these are the members of the Control Robotics Group, my official "home", especially Chris Terman and Steve Ward (now Professor Ward). The members and ex-members of the Programming Technology Group must also be mentioned, especially the chief implementors of the language MUDDLE: Chris Reeve, Dave Cressey, and Bruce Daniels.

In addition, I would also like to thank Professor Gerald Sussman and Allen Brown of the M.I.T. Artificial Intelligence Laboratory for technical discussion and encouragement.

And of course I must thank my parents, who I think had nearly given up hope that I would ever grow up and finish school.

Artificial academic distinctions to the contrary, a task such as this dissertation is never really "finished"; it is eventually abandoned to stand on its own feet if it can. But I can never abandon what I owe to the persons mentioned above, as well as many others who must go unmentioned for lack of space.

This work was supported in part by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N00014-A-0362-0006.

Table of Contents

Dedication

To a Small Subset of Humanity

gorged with the patterning rhythms of change -
motion!
physical, audible, visual, logical
all responsively guided and guiding
layered in patterns perceived.
human!
borne from the sameness of chaos, eternally;
with their shattering rhythms in flexion
merging to pattern a word:
understand.

11 P.M., Friday, May 5 1974

"...the art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible."

--Edsger W. Dijkstra,
Notes on Structured
Programming

Chapter 1
Introduction

1.1 DALI

A computer's mechanical ability to mindlessly iterate marginally differing calculations, coupled with its speed and ability to control other devices, can be used to create and change visual images -- pictures -- at a very rapid rate. This capability is the cornerstone of the two fields of computer animation and interactive computer graphics. Utilization of this naked computational power requires that it be organized, grouped and ordered in a way that highlights aspects important to human beings, suppresses details they consider unimportant, and most of all provides them with an effective means of thinking about and describing their intentions. In other words, a "high-level" programming language is needed.

This document describes such a language. It is called DALI, for Display Algorithm Language Interpreter, and is designed for the creation and control of changing pictures exhibiting complex inter-element interactions. Embodied in DALI is a uniform, hardware-independent methodology for the description of a wide class of dynamic pictures.

A primary characteristic of DALI is modularity, in the sense that picture elements themselves define their own manner of behavior. This is accomplished by constructing pictures not as assemblages of passive data elements, but rather as structures of active elements akin to processes. These elements, called picture moodules, can contain arbitrary user-written procedures which locally define the behavior of the picture element.

The capabilities of DALI in comparison to more classical schemes for picture manipulation are most readily highlighted by examples. The examples used here, and many others in this document, are taken from computer animation for clarity and dramatic appeal. It should be realized at the outset, however, that DALI is not "just" an animation language as that term is normally used, but rather covers the spectrum from animation to interactive computer graphics and may also have application in computer simulation.

Fig. 1-1a shows a circular image constructible by a program using any reasonable computer graphics system. Many graphics systems allow a program to vary certain specific parameters of such an image, for example, its rotation, translation, and scale. An appropriate program can thus change both the rotation and translation parameters quasi-simultaneously in proportions that give the illusion of a rolling ball. This is shown in Fig. 1-1b.

What is not supported by such systems is the construction of a "ball" image in which rotation and translation are inherently coupled. Simply "telling" this "ball" to move would cause it to "roll" to the designated position, varying its own rotation with its translation in the necessary fashion. The creation of such objects is a basic capability of DALI.

An important aspect of DALI is that objects such as the "ball" are defined by arbitrary user-written programs. Thus, the "ball" could react in motion to its external environment as shown in Fig. 1-1c. There, the only command given the ball was "move". It "knew" that it should go over, not through, bumps; that it should squash itself when it falls off cliffs; and that it shouldn't step on the daisies. Alternatively, the daisies could "know" that they should crush when run over by a ball.

Another aspect of DALI is implicitly illustrated by Fig. 1-1c: only that portion of the DALI "program" needed to move the "ball" is invoked when the "ball" moves. This is not done by arbitrarily dividing the

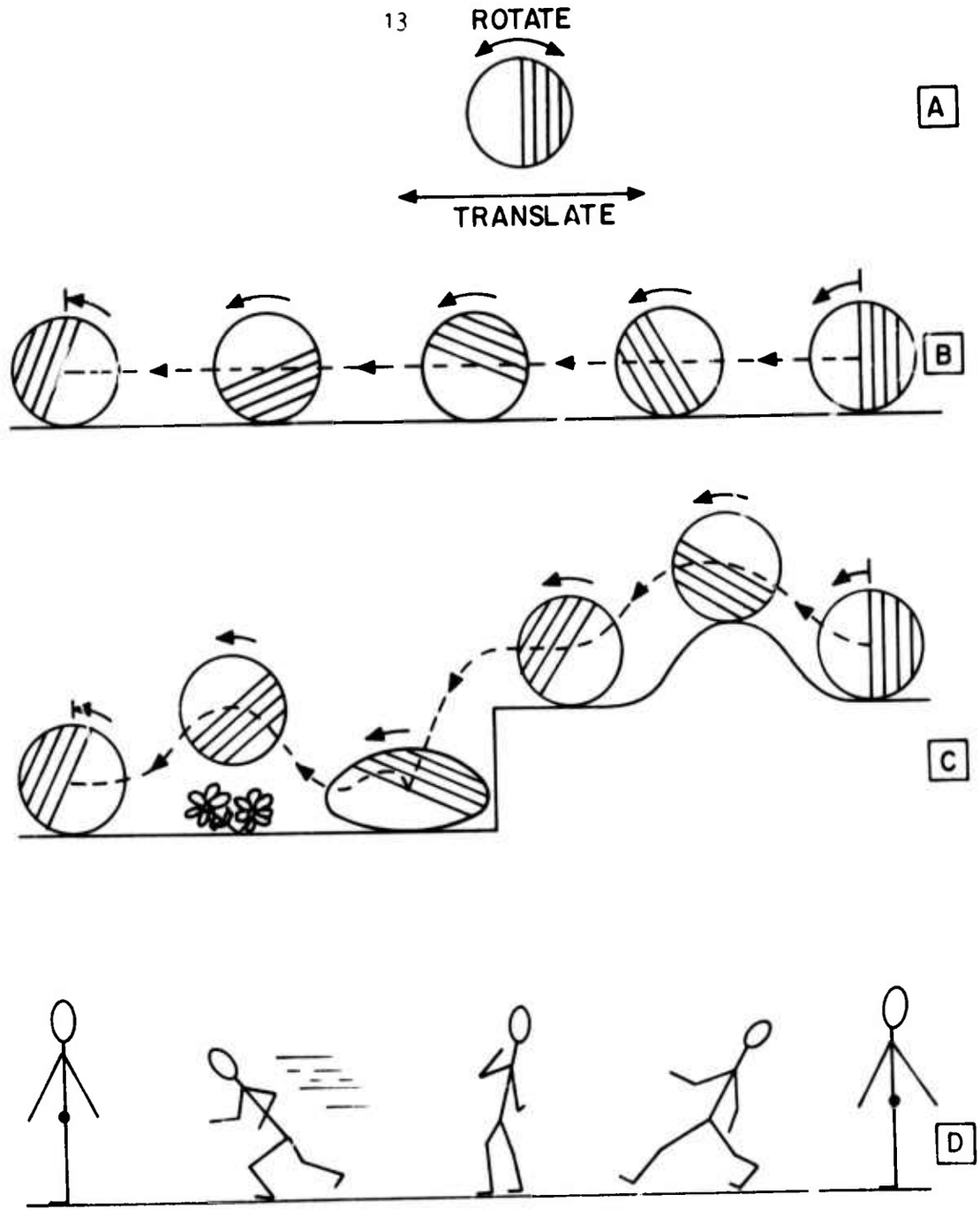


FIGURE 1-1

picture into unchanging background and changing foreground, but rather arises naturally out of low-level picture element interrelationships: individual changes propagate through the picture, affecting only those picture elements involved in the change.

Of course, the image displayed need not be a ball. It could, as in Fig. 1-1d, be a stick figure which walks to the desired place when told to move -- or runs, or jumps, depending on the amount of time it is given. The program giving the command to move can be completely unaware of how the command is carried out; the same command can roll a ball, walk a man, drive a car, etc.

All of the capabilities illustrated above could, with sufficient effort, be programmed in more conventional computer graphics systems, by virtue of the fact that they do, indeed, incorporate a Turing machine. What DALI provides is a general, systematic way of viewing such capabilities, a set of concepts around which to organize one's thinking about changing pictures. By this virtue, DALI lays claim to being a true high-level programming language for dynamic computer graphics, the art of creating changing pictures which lies in the intersection of computer animation and interactive computer graphics.

1.2 Relation to Other Work

A great deal of research in computer graphics deals with conquering the sheer brute load of computation needed to generate pictures with a computer. Examples of this include eliminating hidden lines and surfaces; clipping and "boxing"; performing perspective transformations; generating smooth curves and surfaces; and actually producing a visual image on a cathode-ray tube or other device. Newman and Sproull [New2] describe many results in these and adjacent areas.

Such research can be viewed as an attempt to make a computer into a "super paintbrush", entirely analogous to the more usual view of a computer as a "super adding machine". Even the meaning of "super" is invariant: fast, accurate, flexible, and inexpensive relative to competing technologies. This is a necessary goal. Just as a computer must be a "super adding machine" before issues like process control, artificial intelligence, information systems, etc., can be effectively addressed, it must be a "super paintbrush" before computer animation and interactive computer graphics can be effectively addressed.

The research reported here makes the basic assumption that a true "super paintbrush" exists. This may or may not be true at the present time; quite powerful graphics hardware certainly does exist [E&S1,E&S2], and currently projected decreases in hardware costs promise to bring very powerful "paintbrushes" into more common use. [Spec1]

Given that assumption, the question becomes how we make use of such "super" capabilities. One place to begin is by adding to the basic capabilities more powerful means of control over what is drawn, hopefully increasing our effective ability to create changing pictures far beyond historical manual capabilities. To achieve this control, an ability to describe the picture-which-changes in a computationally effective manner must be devised. That is the goal of the reported work.

Work directly related to that reported here falls into three areas: (1) subroutine packages for general computer graphics, e.g., [E&S1, Tho1, New2 Chap. 5 and 8, Rul1]; (2) programming languages and extensions for general computer graphics, e.g., [Hur1, Chr1, Sm.1, New1]; and (3) programming languages for computer animation, e.g., [Kno1, Bae1]. All of these have the common aim of creating a visible image by communicating data and commands to a display device from a program. Many also provide some mechanisms for communicating interactive graphical input to a program, but this aspect will not be

emphasized in this research. Our interest is further restricted to those aspects of graphics systems related to changing created pictures, as needed in interactive computer graphics and animation. In that area, nearly all graphics systems share a common aspect for which DALI offers an alternative, namely the use of what we shall call an instance tree (defined below) to represent the picture. Therefore, rather than discuss many individual efforts in what would be a rather repetitive fashion, the general rationale behind instance trees, their advantages and their disadvantages, will be discussed; then, exceptions to the use of instance trees will be covered.

The almost universal choice as a method for changing a displayed image -- with some exceptions, including EULER-G [New1] and SKETCHPAD [Sut1], which will be discussed -- is to allow the user to modify an internal data structure called the display file. The display file is a complete description of the desired visual image, in the sense that it contains all the data scanned to produce the signals which drive the visual-image-producing hardware; the scanning process may be performed either by hardware or by software, or by a mixture of the two. A result of this use of a display file is that from a program's point of view, the display file is the picture: changes to the display file are changes to the picture, and the structure of the display file is the natural structure of the picture.

There is surprising unanimity -- again, with a maverick, BEFLIX [Kno1], to be discussed -- as to how the display file should be structured, so much so that the principal differences between structured display file systems lie primarily in the choice and syntax of primitives, not in the structure itself: The display file is invariably structured as a reentrant tree, reminiscent of a program with subroutines; this is the instance tree mentioned above. The manner in which instance trees are typically used to represent a picture is described in some detail below.

The visible image is created from the instance tree in the manner of a processor executing a program: the tree is scanned in a depth-first manner starting at the top node, and the immediate descendants of each node are "called" like "subroutines" of their parent node. A given node may have several parent nodes due to re-entrance; such a multiply-used node will be "called" several times and thus the "subpicture" it represents will appear in the final picture several times. This instance-tree scan may be performed directly by special-purpose display hardware [E&S1, E&S2], or it may be done at least partly by software.

The terminal nodes of the instance tree contain data describing one or more primitive visible objects, e.g., dots and lines; these terminal nodes are often referred to as items, and are similar to procedures which call only primitive (built-in) procedures. The non-terminal nodes of an instance tree are often called groups. Groups contain only references (pointers) to items and to other groups; circular group references are disallowed, since there is no way to terminate the implied recursion. The references which groups contain are called instances of the objects referred to; the object referred to by an instance is called its master.

The relationship between an instance and its master is very like that between a procedure call and the procedure itself; this analogy is made even closer by the fact that instances commonly have parameter settings associated with them, somewhat like procedure arguments. Unlike general procedure arguments, however, the set of possible instance parameters is fixed: the way a given instance parameter can affect the display of its master is fixed by the implementation, and only those parameters provided by the system as primitives can be used. The types of parameters typically provided for use in instances can be somewhat arbitrarily divided into two groups, here called transformations and attributes.

Transformations are analytically definable two- or three-dimensional mappings between coordinate systems. Typically they include

translation, scaling (zooming), rotation, and clipping; the latter is the blanking of any part of a visible object which would appear outside of a given polygonal area, usually rectangular. Successive transformations are concatenated, i.e.: For each use (instance) of an item (set of primitive visible objects) there is a distinct directed path from the root of the tree to the item, passing through one or more instances; the total transformation applied at each use of the data in the item is the concatenation of all the transformations along that path, performed in the order indicated by the path's direction. Since several such paths can exist, a given item can simultaneously produce several images differing in position, size, orientation, etc. Exactly how such concatenations are performed is described in [New2].

Attributes are somewhat less analytically tractable variations on a master. They include such things as color, intensity, "auto-blink", "hit" sensitivity, etc. The manner in which they are concatenated, if they are concatenated, varies widely.

Typical operations which can be performed on instance trees include: (1) the creation of items, involving the specification of the primitive elements to be included in the item; (2) the destruction of items; (3) the creation and destruction of groups; (4) the insertion of instances into groups and the removal of instances from groups, accompanied by, respectively, the creation and the destruction of the instances involved; (5) the modification of instance parameters; and, less often, (6) the insertion and removal of primitive elements from items. Smith's GPL/I [Smi1] is a good example of a language system incorporating nearly every feasible primitive; it also has a very nice syntax for group and item construction.

Many minor variations on the above-described instance tree structure exist. For example, many subsets of the instance parameters listed are used; item data (primitives) can sometimes be included as part of a group; and "instance parameters" sometimes appear as part of a master rather than as part of an instance, strange though that may seem.

Instance tree display file structures are in common use, for example in [E&S1, Tho1, New2, Rul1, Hur1, Smi1, Chr1, Bae1]. Significantly, this type of structure forms the basis of the recently proposed protocol [Spr2] for computer graphics across the ARPA network, a communications net connecting many different types of computing facilities. Furthermore, the only currently published book attempting to deal with interactive graphics as a whole, [New2], refers to an instance tree structure as simply a "structured display file", implying that only this one type of structure is worthy of consideration! Clearly, instance tree structures must have advantages; what are they?

In many cases, a particular instance tree structure can be chosen that is extremely close to the capabilities of the display hardware, so that it can be directly used as hardware input. Being able to make such a match is extremely valuable, since it has great speed advantages especially when refreshed displays which continually "re-execute" the structure are used [E&S1, E&S2, Pfi1, Wat1]: the displayed image immediately changes with changes to the structure. However, instance trees which do not match the abilities of the display hardware are often used, as must be the case with the ARPA network protocol. Not all display hardware has subroutining, for example, and general rotation is rather uncommon. For subroutining, host computer interrupts can be used to simulate more powerful display hardware, as in the system described in [SUW1]; but for rotation and most transformations, such simulation is impossible. In the latter case, a second transformed display file must be created from the instance tree [New2], containing multiple copies of masters which are used with different instance transformations. This clearly vitiates the speed advantage, so there must be other advantages.

An advantage of instance trees often mentioned is space saving due to multiple use of instances [New1]. This is very close to the arguments originally made for including facilities for subroutine or procedure use in hardware and programming languages. Like the multiple-procedure-use argument, the multiple-instance-use argument is not the

full story. The only substantial use of repetition in instance trees is at the lowest level -- sharing of immediately displayable items, particularly characters. Furthermore, much space-saving of this sort is illusory. Especially when many transformations are available, instance overhead is high enough, and immediate multiple-line or -dot formats are compact enough, to wipe out the prospective space saving. For example, it usually does not save space to use an "arrow" instance in both a "diode" and a "transistor". Thus space-saving alone would require only a one-level tree. Yet trees of depth 4 or 5 are common -- seldom more, as pictures tend to be much "broader" than they are "deep". Furthermore, if a transformed display file is necessary, much space saving of this sort is intrinsically wiped out by the multiple transformed copies needed. So another advantage must exist.

The primary advantage of instance trees, a conceptual advantage not dependent on hardware, is this:

> Instances provide <u>loci of control</u> over whole sections of the picture, because change to instance parameters propagates down the instance tree to its leaves.

A single change to a rotation parameter, for example, can cause hundreds of changes to individual lines; elements of a group can be moved relative to one another, or moved in parallel maintaining relative positions, by use of a two-level instance structure. This automatic propagation of change is a major conceptual advantage which greatly simplifies the control of a changeable picture. Such propagation, and not just the naked ability to create and name groupings of objects, is what allows the user to think in terms of "higher-level entities".

However, the kinds of changes which can be propagated by instance trees are limited to a fixed set of transformations and attributes, and the path of propagation is fixed: parameters can depend only on like parameters, and only those of the parental node. Thus, for example, it is not possible to make the position of text labelling part of a

rotating object depend on the object's rotation and shape so as to keep the text unobscured. Another example: a scale change cannot cause the amount of visible detail to vary; no predefined "level of detail" attribute exists in any instance tree system known to the author.

Therefore, instance trees cannot fully represent a dynamic picture, in the sense that they cannot produce an arbitrary desired picture which is fully definable as a function of a set of parameter values. This, it must be noted, is a problem distinct from that of representing static pictures for purposes of display. The latter is a significant problem only if the picture is so complex that space is a major issue, or if substantial transformations, such as hidden-surface removal, must be applied to the picture as a whole.

That an instance tree cannot fully represent a desired picture is clearly not an insuperable disadvantage. Dynamic graphics systems are generally embedded in general-purpose computing systems, so a program can always be written to bludgeon the instance tree into "being" the right picture, effectively providing the special-purpose parameters and relationships among picture elements which the instance tree cannot provide. To do this, a separate data structure is needed to relate elements of the instance tree with each other and with the real data they are representing. Not surprisingly, such data structures can be extremely complex; this is reflected in the wide literature on "data structures in computer graphics", e.g., [vDa1, vDa2, Abr1, Wil1, Gra1, Cot1], and on the fact that computer graphics pioneered the use of many of the most complex types of data structures -- e.g., rings in SKETCHPAD [Sut1] and CORAL [Suw2], and associative data bases in LEAP [Rov1]. This need for complex data structures must be considered a major problem, if for no other reason than that the data structures themselves are considered a major problem.

One system not using an instance tree as a picture representation is Knowlton's BEFLIX [Kno1], one of the first, and still one of the most

successful, systems for producing computer-generated movies; examples of movies produced with BEFLIX include [Bel1, Kno2]. BEFLIX conceived of the picture as a two-dimensional array of intensity values called a surface. One "fine" (184 x 152) or two "coarse" (92 x 126) resolution surfaces were available, operated on by a set of scanners which, with a "scanner language", could be made to move around the surface(s), read and change intensity values, and communicate with one another. In addition, a "movie language" actually produced film output and performed operations on rectangular areas in surfaces such as copying other areas, dissolving to other areas, zooming, etc. The use of structures like an instance tree is actually orthogonal to BEFLIX's use of surfaces; a tree of surfaces and instances of surfaces could be constructed to fulfill the same needs. However, the small number of surfaces available (two) -- further restricted to only one surface in a more recent similar system, EXPLOR [Kno3] -- prohibits this. The user must always consider his picture as a pure image, and not, e.g., as a collection of independently existing but related objects which he can manipulate and alter; this may have advantages in some purely artistic endeavors, but is a decided disadvantage in more general use.

Another system departing from the instance tree syndrome is Newman's EULER-G [New1], an extension of EULER [Wir1], a language which is itself a generalization of ALGOL. In EULER-G, execution of a frame procedure causes immediate construction of a monolithic transformed display file which is directly digestible by the display hardware. The instance tree actually has virtual existence during this execution, as described below:

The EULER-G system interprets primitives such as

line to [x,y]

which draws a visible line from the "current" position to the position (x,y), according to the current settings of transformation parameters. The current transformation can be concatenated with other

transformations in a dynamic block-structured fashion as part of a call to a general EULER procedure; at the return from such a call, the transformation in force before the call is restored. Thus, the virtual tree of procedure calls performs the functions of an instance tree but allows more generality, since an isomorphism is attained between a picture element and the arbitrary procedure used to create that picture element: The contents and structure of the virtual instance tree are controllable in a natural way by the standard EULER mechanisms for parameter passing, iteration, and conditional execution. Thus, the picture can in fact be an arbitrary function of a set of parameters; this is a capability which, as was pointed out above, is practically impossible to attain with instance trees -- or, indeed, with any other passive data structure. However, this ability is obtained at the expense of completely re-creating large portions of the picture both in order to make changes, and also in order to process pointing inputs from graphic input devices. The resultant execution-time overhead is significant, and "smooth" motion is essentially impossible.

Another general approach to creating changing pictures is what is referred to in [New3] as the "viewing algorithm" approach. The term viewing process approach will be used here instead, both because "viewing algorithm" is used with a different meaning in [New2], and because this approach conceptually involves the use of two processors sharing a single physical memory. A single physical processor may of course be time-shared to achieve the same effect, and so we will speak of two processes as being involved; these will be called the application process and the viewing process.

The application process performs the "real" work involved, manipulating the data in the shared memory in the course of whatever calculation is being performed. While the application process is running, the viewing process simultaneously accesses the data being manipulated, repeatedly and continuously traversing all the data to be

displayed and constructing a picture which represents that data in the desired fashion. Since the display process' operations are arbitrarily programmable, this method shares with EULER-G the advantage that the structure and parameters of the picture can be very general.

In a certain sense, the viewing process method is utilized by any system which uses a separate process or processor to do the continuous re-display needed with refreshed image generation hardware. What distinguishes the viewing process method is: (1) the lack of an internal representation of the picture distinct from the structure the application process uses; and (2) the fact that the viewing process independently accesses the shared data structure to continuously regenerate the picture while the shared data is simultaneously being altered by the application process. If condition (1) is not met, the system is usually closer to an instance tree system; if condition (2) is not met, the system is instead closer to EULER-G.

Because the viewing process accesses the shared data structure while it is being updated, the viewing process method has the advantage, in theory, that the application process need not take note of the fact that a picture is being generated; it merely changes its data and the picture automatically changes in response.

In practice, however, a truly formidable amount of inter-process synchronization is necessary because changes to the data can leave it in a momentarily inconsistent state capable of thoroughly deranging the operation of the viewing process. For example, it is not possible with conventional computer architectures to add an element to a two-way linked list without, at some point in the operation, making the pointer relations "incorrect", i.e., at some point the structure is not in fact a two-way linked list; if the viewing process depends on the structure's always being a two-way linked list, it may fail.

Another problem, which this method shares with EULER-G, is that the overhead involved in completely regenerating the picture can be quite high. If a single time-shared processor is used, this will slow the

application process considerably. If, on the other hand, two physical processors are used, the application process may run too fast for the viewing process to keep up. This may lead to pictures which are inconsistent in the sense that different parts of the picture reflect differing successive states of the data; such global inconsistencies may also lead to failure of the viewing process if considerable care and forethought are not applied.

The problem of the application process "running too fast" is actually more severe than it might appear at first glance. This is due to two factors: First, a data structure which is appropriate and efficient for the application process may be decidedly inappropriate and inefficient for the purpose of generating the desired image. Second, if the desired picture is not a straightforward transformation of the application data, the amount of computation required to create the desired picture may be quite large. As an extreme example of both of these problems, the viewing process may have to construct an aesthetically pleasing layout of a graph described only by its connectivity matrix.

Despite these problems, the viewing process approach has been successfully used in several applications where the desired picture was a simple transformation of the data and a data structure could be chosen which was a good compromise between the needs of the application process and the needs of the viewing process; examples are [Chr1, Rob1, Sut1]. Of particular note among these is Ivan Sutherland's SKETCHPAD system [Sut1], which will now be discussed at some length.

SKETCHPAD is of course the seminal work that pointed out the virtues of interacting with a computer by means of pictures; the historical effect of this system on the entire field of computer graphics would be difficult to over-emphasize.

In discussing SKETCHPAD, it is important to note that this system is neither a "graphics package" nor a "graphics language" as those terms

are nor mally used, but rather an application program which used interactive computer graphics as an aid to carrying out its application -- namely, the creation of pictures.

The method used to display the pictures described by SKETCHPAD's application data -- i.e., a viewing process -- has already been described. Of interest here are the techniques used in the application itself.

It must first be pointed out that SKETCHPAD's application should not a priori be overly relevant to the general topic under discussion, since that application was the creation of static, not dynamic, pictures; the motion of pictures visible in SKETCHPAD was due to the use of the display process technique to show successive intermediate stages in the construction of the desired static picture. What makes SKETCHPAD relevant to this discussion is the fact that the method it uses to construct a desired static pictures is modification of a pre-existing, and possibly null, picture; and such modification is, of course, change.

Keeping this in mind, we may note that SKETCHPAD's application subsystem, but not its viewing process display subsystem, structures the picture in two ways: First, the picture is structured as a re-entrant tree composed of instances of master subpictures; the terms "instance" and "master" originated with SKETCHPAD. Second, unlike the unidirectional change propagation of the "instance tree" systems discussed earlier, the SKETCHPAD picture contains an arbitrary non-hierarchical non-directed graph composed of multi-way constraints, i.e., N-ary relations between the coordinate and other values used to define the picture drawn by the viewing process. One of the primary jobs of the application subsystem is to make every value used in the picture reflect all the relations between values desired by the user. If no circularity exists among all the relations -- i.e., no values are defined in terms of themselves -- appropriate values are found at once; otherwise, every value in the picture is iteratively re-computed, in the manner of Gaussian iteration or relaxation [Ral1, Var1], hopefully

converging to the desired solution. Relaxation was facilitated by defining constraints as functions which, applied to the values constrained, returned an error value.

SKETCHPAD's constraint network is more general than unidirectional instance tree change propagation in three ways: First, values can be functions of other unlike values; for example, a rotation can depend upon a displacement. Second, since constraints are not directional and relaxation can be used, every value in the system is potentially a locus of control over every other value; thus, by inserting a very "hard" constraint between a value and an input device such as a knob, direct control over any aspect of the picture can be achieved. Third, relationships can be used which are not representable hierarchically -- for example, parallelism.

As with instance tree systems, the relationships available in SKETCHPAD are fixed in the sense that a user cannot define an arbitrary desired constraint without doing programming "behind" the user interface.

However, multi-way constraints appear to be less general than the picture/procedure isomorphism available in EULER-G, since structural changes to the picture -- i.e., the addition and deletion of picture elements, including constraints -- are not readily expressible in terms of error-function constraints. For example, Sutherland states in [Sut1] that it is not possible to create a constraint that causes a corner to become rounded; the reason for this is that such an operation involves the addition of a circular arc and the modification of the constraints used to hold visible lines onto position values.

The principle disadvantages of SKETCHPAD -- or, rather, of a programming language system which could be derived from SKETCHPAD's command language -- are two, and they both involve the use of relaxation.

First, relaxation is a hill-climbing technique, and it is all too easy for any system using such techniques to become "caught" at local

maxima different from the desired maximum -- i.e., to reach states where "you can't get there from here". In the interactive situation of SKETCHPAD, this is not too bad, since a non-malicious human operator can usually help the system over such "humps"; a malicious operator can, of course, always bollix such a system easily. Considered as the only solution technique available in a programming language, however, the use of relaxation is quite limiting.

The second disadvantage concerns the efficiency of the system. The most important consideration here is that the apparent inefficiency of SKETCHPAD is not so much implicit in the system itself as imputed from the manner in which it was used. With the exception only of the multiple-truss bridge examples, there are no examples in [Sut1] which really "need" relaxation. However, constructing many of the examples without relational circularity generally requires some forethought, planning, and a more detailed knowledge of the system; in general, doing this is less convenient for the user, and user convenience was a major goal of SKETCHPAD. It must be noted, however, that the forethought and planning required are well within the range of that normally required for programming, so the "need" for "inefficient" relaxation must not be considered a disadvantage in SKETCHPAD considered as a programming language.

In summary, it can be said that SKETCHPAD contains the seeds of a graphics programming system that is more powerful and not intrinsically less efficient than the vast majority of graphics systems currently in use. However, due at least partially to the self-referential nature of its application, SKETCHPAD can, if viewed hurriedly, engender much confusion over the difference between static and dynamic pictures and the nature of non-hierarchical structure in pictures; of course, no such issues were issues until SKETCHPAD existed.

Perhaps surprisingly, the majority of work in computer animation impacts only obliquely on the work reported here. This is the case

because work in computer animation has been confined to two areas which are "lower level" -- in the sense of "high" and "low" level languages -- than the area of the reported research.

This first of these areas is the creation of a "super paintbrush" capable of rapidly producing pictures at least comparable to those produced by conventional animation techniques; Baecker [Bae1] discusses animation work in this area. As was mentioned earlier in this section, a "super paintbrush" of some sort is assumed in this document.

The second area of animation research is the detailed synthesis of the complex interrelated motions and sequences of structural change which are required in animation; examples are [Bae1, Bur1, Par1]. The data gathering, functional parameterization, synchronization, etc., which are necessary for animation impinges on the reported work only to the extent of placing certain general requirements on the system developed: (1) it should be possible to make picture parameters arbitrary functions of time; (2) temporally parallel changes to many parameters should be possible; and (3) facilities for temporal synchronization and coordination should exist. All of these criteria are met in DALI.

Baecker's proposed "Animation and Picture Processing Language" (APPL) [Bae1] is, however, relevant, and will now be discussed.

In addition to providing facilities for parallel change, synchronization, and the gathering of pictorial and motion data, APPL provides a purely hierarchical structure for pictures. This structure is extensible in the following sense: Executing an APPL statement such as "MOVE picture BY distance" -- meaning translate a (sub-) picture by a given amount -- recursively applies MOVE to all the subpictures of the picture until either: (1) a primitive object, e.g., a dot, is found to which a built-in MOVE is applicable; or (2) a user-defined "picture type" is found for which the user has provided a MOVE primitive. Thus change is propagated in a downward hierarchical manner, as is the case

with instance trees; but more flexibility is available than with instance trees, since by defining new picture types, the user can arbitrarily vary both the effect of a given command and the manner in which it is hierarchically propagated.

However, APPL provides no non-hierarchical propagation of change whatsoever: even a "relative line" -- i.e., a line segment defined by an endpoint and a vector distance, and "automatically" moved when the endpoint is moved -- cannot, as Baecker points out, be defined in APPL. Why this should be the case, and in what sense it is true, is perhaps best explained by example:

Suppose there is a picture PA containing an endpoint E, and we wish an otherwise unrelated picture PB to be centered at E. Initially, PB is constructed using E, and shares it with PA. Now, if PA is MOVEd, the position of E will -- or at least can -- change; but there is no way to indicate that since E changed, PB should be MOVEd also. The only way to make PB change whenever E changes is to make PB part of PA. But there are situations where this "containment" solution does not work; for example, a line which is to join the centers of two independently moving objects cannot be "part of" either object. Maintaining such non-hierarchical relationships is referred to as "constraint satisfaction" in APPL and considered outside of the domain of the language.

It should be noted, however, that none of the examples given above -- the relative line, the centered picture, and the line joining centers -- involve circular constraint relationships; the desired picture is quickly and easily computable in closed form, without the use of any type of "relaxation". The problem is that the lack of any structure other than hierarchical structure makes it impossible to express the desired behavior of the picture without stepping outside the bounds of the formalism which is provided.

The problem that purely hierarchical picture description formalisms have significant practical limitations also affects instance tree systems. There, however, this problem is overshadowed by other

concurrent limitations, including the necessity of having parameters depend on like parameters and the limited set of parameters available. It is also the case that instance tree systems generally provide some limited, but useful, non-hierarchical relations, e.g., relative lines. Since APPL removes the additional limitations and provides no non-hierarchical relations at all, it is an excellent example of both the capabilities and the limitations of purely hierarchical picture description methods.

In comparison, a major goal of the work presented here has been to design a system in which arbitrary user-defined non-hierarchical relationships -- as well as hierarchical relationships -- are specifiable in a manner conducive to their efficient computation.

1.3 Summary

Chapter 2 presents a characterization of dynamic computer graphics and the basic system and language organization of DALI. In particular, it presents DALI's division into (1) M-DALI, dealing with changes that occur instantaneously, i.e., at monadic instants of time; and (2) S-DALI, a superset of M-DALI, which deals with smooth changes occuring across temporal intervals as sequences of monadic changes.

Chapter 3 presents the basic concepts of M-DALI, including: the four basic DALI objects, namely outputs, daemons, picture modules, and picture functions; the two structures which thread through a picture, namely the containment tree and the data web; and the manner in which DALI pictures, as programs, are "executed". These are the central issues in DALI. Understanding this chapter is absolutely critical in understanding DALI.

Chapter 4 discusses further issues in M-DALI, including coordinate system transformations, structural changes to picture elements

interdependence, deletion, and "hit"-testing; it also includes a realistically large example.

Chapter 5 considers the problem of circular dependence among daemons, leading to iteration and relaxation.

Chapter 6 then presents S-DALI, and Chapter 7 presents conclusions and summary.

Appendix 1 provides an alphabetized list of the primitive DALI procedures and summarizes their actions. Appendix 2 similarly lists the objects defined by DALI, their components and purposes. Appendix 3 presents some details concerning the possibility of retentive storage management -- "garbage collection" -- not covered in the text.

Chapter 2
A Global Overview of DALI

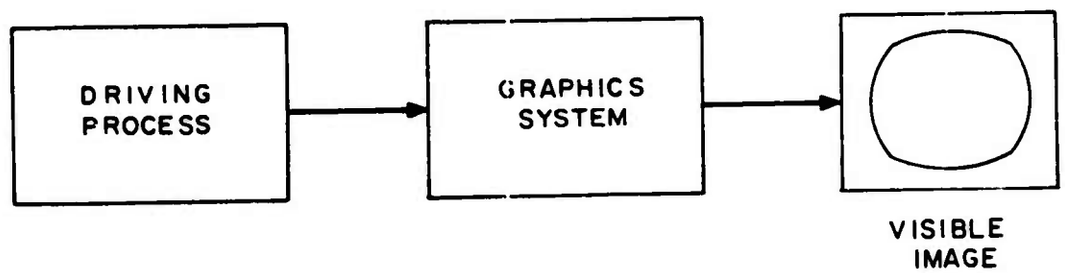
2.1 Static Versus Dynamic Translation to Pictorial Form

At the highest level of abstraction, all dynamic graphics systems including DALI are divisible into two primary components: the driving process and the graphics system. (Fig. 2-1a)

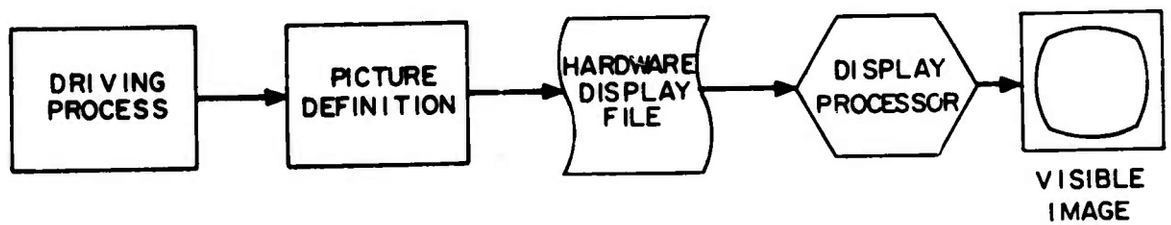
The driving process is the ultimate source of the data on which the picture is based. It is the applications program in execution, and contains in a form which it finds convenient the data which is to be interpreted graphically. This data may or may not have explicitly graphical components. It could be an array of numbers, to be shown as a graph; a complex interconnected data structure, to be shown as a circuit schematic; the positions and attitudes of characters, a background, and a camera angle, to become one or more frames of a movie; etc.

The function of the graphics system is to translate selected parts of the driving process' data into the desired pictorial format, a process which may in general be either static or dynamic. The differences between the two types of translation are the primary subject of this section. They are best explained by first dividing the graphics system a bit further into a picture definition system, a hardware display file, and a display processor (Fig. 2-1b). How this structure relates to the systems described in section 1.2 will be discussed at the end of this section.

The picture definition system translates driving process data into



A



B

FIGURE 2-1 A GRAPHICS SYSTEM

a representation of the desired picture which can be consumed by the display hardware. This representation is the hardware display file; it is accessed by the (hardware) display processor to produce a visible image. We are concerned only with the creation and manipulation of the hardware display file by the picture definition system.

In static translation to pictorial form, the picture definition system always scans all the driving process data needed to produce the whole hardware display file. I.e., if the driving process changes some relevant datum and an updated picture is desired, the picture definition system re-scans all the data for the picture and creates an entirely new hardware display file which replaces the previous one. This process may be viewed as successive retranslation or picture regeneration.

The contrast between static and dynamic translation occurs when the driving process changes the data on which an existing picture is based. The dynamic method does not create an entire new display file but rather alters the existing file to reflect the changes, accessing only that driving process data relevant to the changes. Dynamic translation may be viewed as propagating driving process changes across to the corresponding parts of the hardware display file, bearing in mind that this propagation is an active process which transforms the nature of the change. It may also be viewed as incremental translation or true picture change, as opposed to regeneration.

In the static case, the picture definition component need retain no state information; it is simply one or more pure procedures re-applied from scratch to the driving process data. In the dynamic case, state information must be retained from change to change in order to define the mapping from the driving process data to the corresponding elements of the hardware display file.

Dynamic translation potentially requires less computation than static translation since full regeneration of the hardware display file is unnecessary; the time savings resulting from this can be quite large, especially when few elements of the picture move simultaneously, as is

commonly the case. However, there may be significant overhead involved in updating the state data and actually using it to find out what to change; also, the space needed for the required mapping data may be large.

In addition to a potential saving of computation time, dynamic translation is inherently more powerful than static translation, since the former can produce changes to the driving process changes themselves and the latter cannot. For example, suppose that a driving process datum which is translated into a bar graph element's height is changed from 3.0 to 4.5. All that static translation can do is first show the "3.0" height, and then -- in the "next" picture -- the "4.5" height; it can do nothing else, since the "3.0" height is no longer available when the "4.5" height is generated. In comparison, both the initial and final heights are potentially available simultaneously in dynamic translation; hence, the sudden change can be converted into a smooth one by interpolating the height change into many small successive steps. Lest the reader consider this a frivolity, it should be noted that such smoothing has the very significant effect on the human viewer of maintaining the identity of the changed picture element across and through picture changes. This can be very important if major changes are made in the picture, since it minimizes the problem of discovering how the new state of the picture, and hence of the driving program data, relates to the previous one.

Whether dynamic translation has any inherent power over static translation beyond smoothing and lower computation cost depends on how strictly the line is drawn between the two types of translation. Where this line is drawn is often a question of which data belongs to the driving process, and which belongs to the picture definition as part of its state. For example, the inverse of smoothing, i.e., updating the picture only every Nth driving process data change, requires a modulo N counter. If that counter is considered part of the driving process, i.e., the driving process informs the picture definition only of every

Nth change, then static translation can provide this feature. On the other hand, if the counter is part of the picture definition, i.e., the driving process informs the picture definition of all the changes and the picture definition itself chooses when to run, then dynamic translation is being used. In general it can be said that dynamic translation provides more convenient mechanisms than static translation for relating the dynamics of the visual image to the dynamics of the driving program, and in keeping purely display-oriented operations separated from the application.

However, a strict distinction between static and dynamic translation is often difficult to make, since for efficiency reasons, static translation is seldom found in a completely pure form; usually there is some mechanism for maintaining several simultaneously visible display files, so that large quantities of static background data do not have to be regenerated for every change. From the discussion above, this appears to be dynamic translation since only "relevant" parts of the picture are changed; but the mapping between the driving process data and the elements of the picture is sufficiently crude that the overall tone of such as system is really that of static translation. EULER-G [New1] is a good example of system like this.

The instance tree graphics systems discussed in section 1.2 provide for the breakdown of the hardware display file into a large enough number of small units to allow true dynamic translation, possibly with intervening translation into a transformed display file truly corresponding to the hardware display file.

The detailed relationship between the four-element organization presented here -- driving process, picture definition, hardware display file, and display processor -- and the instance tree systems discussed in section 1.2 depends on the capabilities of the display processor. If the display processor is capable of producing a visible image directly from the instance tree used, then the hardware display file is the

instance tree, and the picture definition is the combination of (1) the utility routines provided for manipulating the instance tree, (2) the user programs concerned with manipulating the instance tree, and (3) the user data structure used to relate the instance tree with the driving process' data. In this case, the user data structure comprises the state information of the picture definition. Referring again to the display processor, if it is incapable of using the instance tree directly, then the hardware display file is the transformed display file referred to in section 1.2, and the instance tree is an additional part of the picture definition's state data.

The four-element division of this section may be interpreted for the viewing process method described in section 1.2 in either of two ways: In the first interpretation, the viewing process is considered a complex display processor; here, the display file and the picture definition are intermingled with the driving process' data. As an alternative interpretation, the viewing process itself, including whatever temporary internal state data it may have, is the picture definition; in this case, the hardware display file has only virtual existence as control words and data passed to a simple hardware display controller. The first view is usually more appropriate, since some elements of picture description are usually necessary in the shared data base used by this method. Interestingly, viewing process systems are very often static translation systems which -- when they work properly as in SKETCHPAD -- operate very quickly indeed, injecting no dynamics of their own but constructing new pictures fast enough to show every twitch of the driving process.

2.2 The Top-Level Operation of DALI

DALI fits into the above scheme as a picture definition system for dynamic translation of changing data into pictorial form. It composes the picture definition out of "process-like" elements, called picture modules, and provides for the needed state/mapping data both through state components in the picture modules and through the structure in which the picture modules are embedded.

The notion of dynamic translation as the propagation of computed change is deeply embedded into DALI; it is the basis for the transfer of control from picture module to picture module and from the driving process to the picture modules. This transfer of control comprises the top level operation of DALI, and it proceeds in this manner:

(1) Initially, control resides in the driving process. This goes about its business in whatever manner it finds appropriate, given its programming and data, until (2) occurs.

(2) When the driving process makes a change to data which determines the current picture, this change is detected by one or more picture modules and control leaves the driving process and enters the picture definition.

(3) One at a time, the picture definition's picture modules which have detected change are given control and perform whatever processing they desire. They may directly change the hardware display file; or they may make data changes which are detected by other picture modules, thus causing the latter to eventually obtain control.

(4) When every picture module which has detected change has been run, control returns to the driving process which then takes up where it left off.

As an example of this process, consider the simple problem, mentioned in the preceding section, of making a bar graph element's

height follow a datum in the driving process. The type of picture desired is shown in Fig. 2-2a, and a possible picture definition is diagrammed in Fig. 2-2b. In Fig. 2-2b, small rectangles are "watched" data, circles are picture modules, and dashed arrows point from a datum to the modules detecting changes in that datum. The elements of the picture corresponding to elements of the picture definition have been given corresponding labels: P1 and P2 contain line endpoint data, module R constructs a "relative position", and modules L1, L2, and L3 create and update lines.

When the driving process changes the indicated datum to 4.5, the following happens:

(1) Picture module T notices the change to 4.5, gets control, and using the value 4.5 concocts a new value for P1. This value will be used as the new position of the upper left corner of the bar.

(2) Picture module R, seeing the change in P1, gets control and generates a new P2, the position of the upper right corner of the bar.

(3) The changes in P1 and P2 are noticed by module L1, which updates the hardware display file to contain a line correctly showing the top of the bar, a line between P1 and P2.

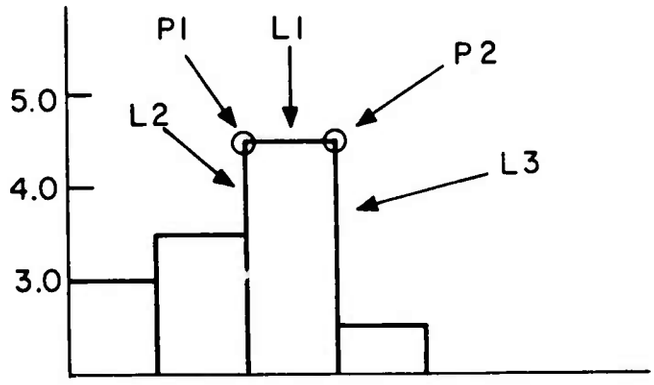
(4) Modules L2 and L3, in a manner similar to L1, update the display file to show correct lines on either side of the bar on seeing changes made to P1 and P2.

(5) Control returns to the driving process.

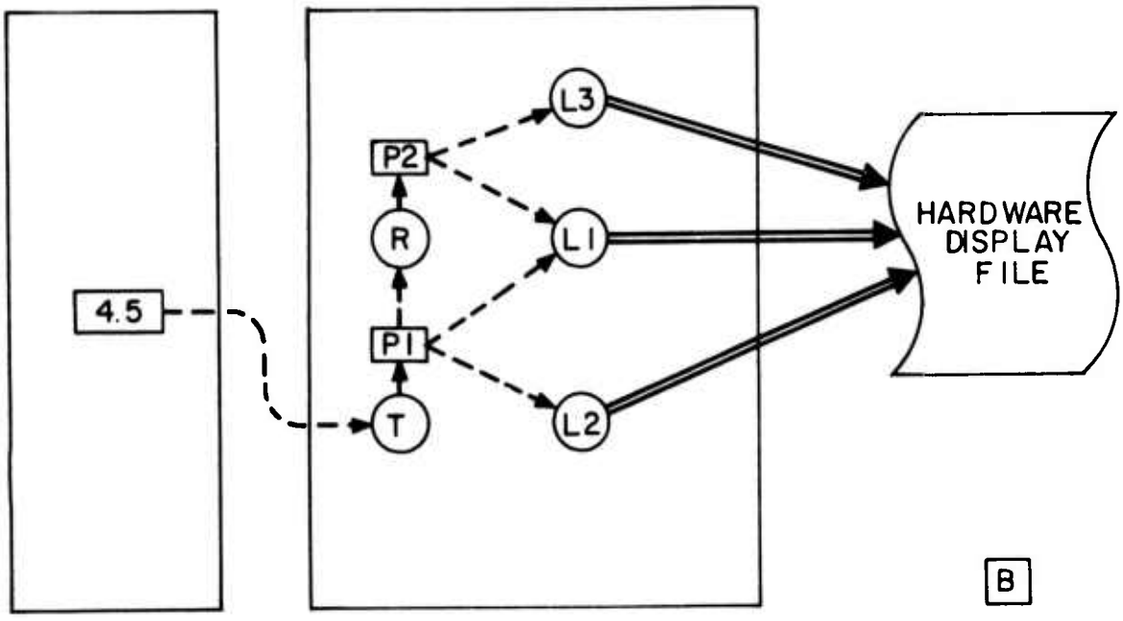
Several points about this example are of interest:

First, the example is oversimplified in that all the modules require data which has not been shown but which is part of the state data. For example, L2 and L3 need the positions of the bottom ends of their lines.

Second, it should be noted that no picture module explicitly "calls" another. T, for instance, does not refer to R, L1, and L2; it just changes P1 and leaves sorting out the result to the DALI system.



A



B

DRIVING PROCESS

PICTURE DEFINITION

FIGURE 2-2 EXAMPLE OF PICTURE CHANGE

Third, we conveniently assumed that R would run before L1, thereby avoiding running L1 twice. In fact, R is guaranteed to run first by scheduling rules built into DALI which oversee the transfer of control from module to module.

Fourth, the mechanism for communication among picture modules -- change a value and let those watching it run -- is the same mechanism used to communicate from the driving process to the picture definition.

2.3 Sudden Versus Smooth Change: M-DALI, S-DALI, and Time

At the normal level of human perception and interest, change to a picture, or indeed to anything, can be viewed as belonging to one of two categories: instantaneous change, occuring at an indivisible, hence monadic, instant of time; and smooth or continuous change which occurs across a temporal interval. This division is reflected in DALI, which is divided into two parts: M-DALI, which deals solely with changes which, though many can occur at once, are all impressed on the picture simultaneously -- at an indivisible, Monadic instant; and S-DALI, an extension of M-DALI, which deals with smooth changes that occur across temporal intervals by treating such changes as temporal Sequences of instantaneous monadic changes. DALI, referred to without any prefixed qualifiers, is actually S-DALI specifically considered as an extension of M-DALI and therefore containing all M-DALI constructs; the term "S-DALI" will often be used to refer to elements of S-DALI which are not in M-DALI.

A primary difference between S-DALI and M-DALI is that M-DALI has no notion whatsoever of time: the only way an M-DALI program can change the picture is to say "make this change now" -- a statement which, since the "now" must always exist and so can be implied, is simply shortened

to "make this change". S-DALI, on the other hand, contains the notion of time as a metric ordering events. Thus an S-DALI program can say "make change C1 at time T1, make change C2 and time T2, and make change C3 at time T3". This would result in the occurrence of changes C1, C2, and C3 in an order defined by the values of T1, T2, and T3; the amount of time elapsing between the performance of the changes is also defined by the values of T1, T2, and T3.

In the preceding paragraphs of this section, the term "time" has actually been used to mean "time as the programmer intends it to pass in the picture as perceived by a viewer", or picture time. This must be distinguished from DALI compute time, the time which necessarily passes while the computation of changes to the picture is taking place.

Picture time is always "frozen" during DALI compute time, i.e., picture time does not change during the computation of a set of picture changes: all processing in DALI is instantaneous with respect to picture time. How this operates in M-DALI may be illustrated as follows:

Suppose an M-DALI program is running to update the picture in response to some driving process operation. Some picture module says "make change C1", and, some period of DALI compute time later, another picture module says "make change C2"; to M-DALI, this whole operation means "make changes C1 and C2 simultaneously", where "simultaneously" specifically refers to picture time. In terms of a motion picture recording of the resultant display, changes C1 and C2 will both be performed during the interval between two successive frames of the movie.

In a situation where the display is being directly viewed while changes are being computed, the system will attempt to make picture-time simultaneous changes appear to be truly simultaneous; but this attempt may be unsuccessful if the system is overloaded.

The numeric values used to indicate times in S-DALI refer to picture times, and are most easily understood as referring to some

number of "movie frames" since the system was reset. Again, if the display is being directly viewed, an attempt will be made to maintain the desired relationship between picture time and real time as perceived by the viewer, but it will not always be successful. How picture time advances will be explained later in this section; for now, it should be noted that picture time remains constant until all the processing affecting a given instant of picture time has been completed.

This strict separation between picture time and DALI compute time is made because the amount of time required for computation can vary in a manner which is quite difficult to predict; and if aesthetically pleasing dynamics are to be created, picture time must be controlled accurately.

There is actually a third time scale involved in the operation of DALI: this is the time the driving process takes to do its own computation, and is called driver time. Driver time is distinguished from DALI compute time because in a purely M-DALI system it serves to separate and sequence successive changes to the picture: the driving program runs, then M-DALI makes a set of "simultaneous" changes, then the driving program runs again, etc. This is something DALI compute time never does; DALI compute time is always invisible relative to the separation and sequencing of picture changes. Driver time has no effect on picture time in S-DALI: picture time is "frozen" while the driving process is running. Driver time may, of course, affect the relationship between picture time and real time.

The manner in which M- and S-DALI interact with the driving process will now be described, with emphasis on the temporal issues involved. At this point it should be recalled that, as mentioned in the previous section, the mechanism by which control is passed from the driving process to the DALI picture definition, and also from picture module to picture module, is by changing some value "watched" by a picture module; this is uniformly the case in both M- and S-DALI.

The interaction between a purely M-DALI picture definition and the driving process consists of repetitions of the following sequence:

(1) The driving process runs for a period of driver time, making a set of changes relevant to the picture. At some point, chosen by the driving process, the current set of changes is presented to the M-DALI picture definition "in parallel" -- i.e., the driving process voluntarily suspends its operation and allows all the picture modules "watching" all the changes to run.

(2) Driver time stops advancing, and computation occurs within the M-DALI picture definition for a period of DALI compute time.

(3) DALI compute time halts, and a set of changes are simultaneously impressed upon the picture.

(4) Control returns to the driving process, and driver time resumes its advance.

Two successive repetitions of this sequence are shown in Fig. 2-3, which illustrates the relationship between driver time. DALI compute time, and "pseudo picture time"; the latter is the sequencing of sets of simultaneous picture changes which derives from driver time and DALI compute time when only M-DALI is used. Such a system is essentially useless for serious animation work due to the circumstantial nature of the intervals between picture changes; however, it can be useful in interactive graphics since close control over picture time is unnecessary in such applications, and the added overhead needed to achieve that control is disadvantageous.

In comparison, the interaction between an S-DALI picture definition and the driving process specifically refers to picture time:

(1) The driving process runs for a period of driver time and eventually presents a set of changes to the picture definition "in parallel" as in M-DALI. At this point, picture time has some value, initially 0.

(2) Driver time stops advancing.

(2.1) Computation occurs within the picture definition for a

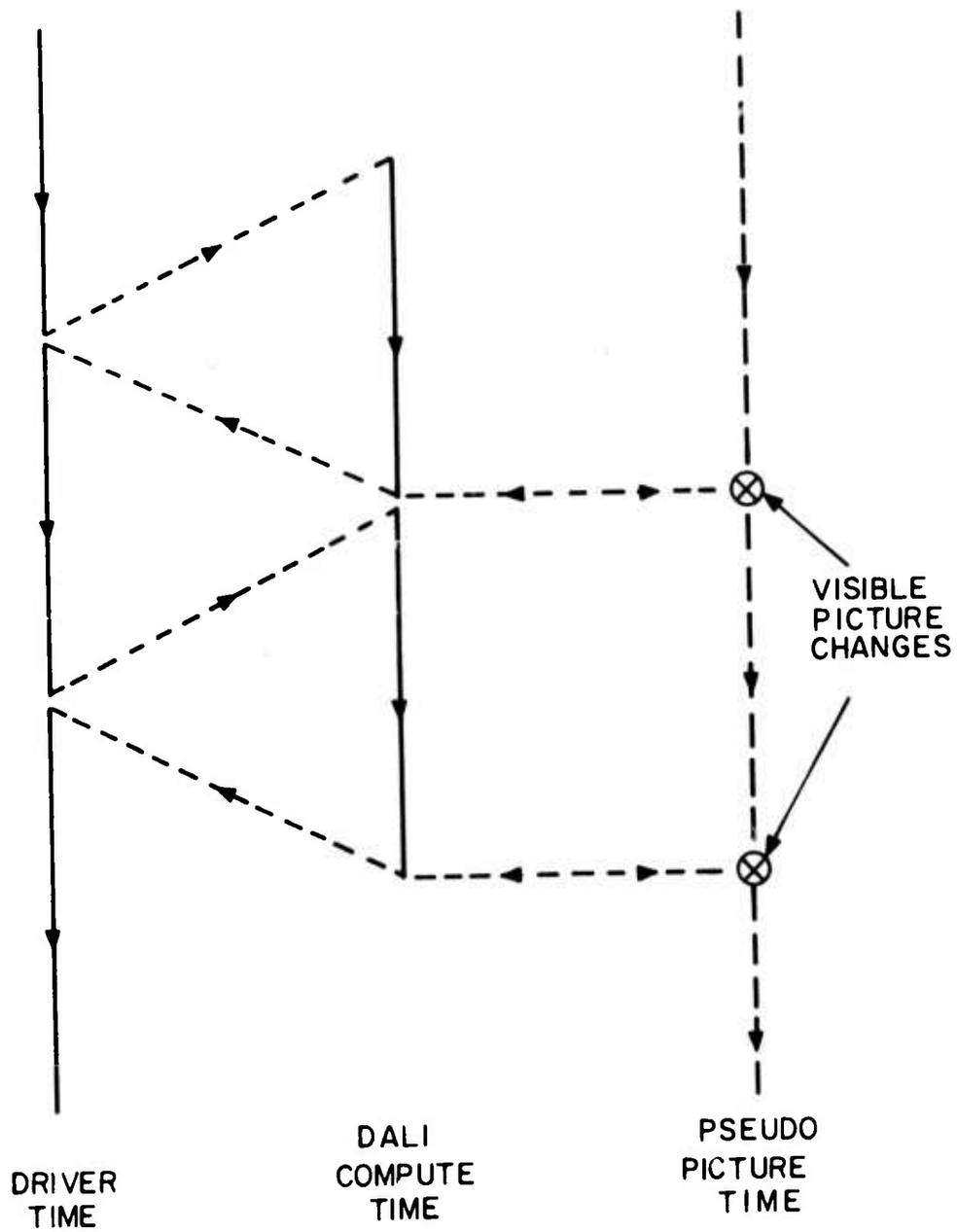


FIGURE 2-3 M-DALI OPERATION

period of DALI compute time, until all the picture modules which "want" to run have been run.

(2.2) If any previous DALI computation created picture changes slated to occur at the current picture time, all such changes are simultaneously impressed on the picture.

(2.3) If any previous DALI computation has scheduled any operations -- not just picture changes -- for a picture time in advance of the current picture time, picture time is advanced to the earliest such future picture time and step (2.1) is re-entered to perform the desired operations; otherwise, step (3) is entered.

(3) The driving process continues from where it stopped, and driver time resumes its advance.

Two repetitions of this interaction are illustrated in Fig. 2-4, which has been constructed under the assumption that picture-time intervals between picture changes are to be constant.

With respect to the preceding S-DALI operation sequence, some additional explanation is needed concerning the scheduling of operations for future picture times which is referred to in step 2.3.

S-DALI programs in fact have a general ability to say "do this at that time", where "that time" is any picture time in advance of the current picture time, and "this" is an arbitrary fragment of a program; such operations are then performed during an iteration of step 2.1 as indicated above.

An operation which is particularly relevant and simply performed at a future time is changing the value of some "watched" datum; this is particularly useful since it can call a multitude of other picture modules into action. S-DALI therefore contains facilities for conveniently performing such "future changes"; in particular, S-DALI contains facilities for creating, as units, finite sequences of future

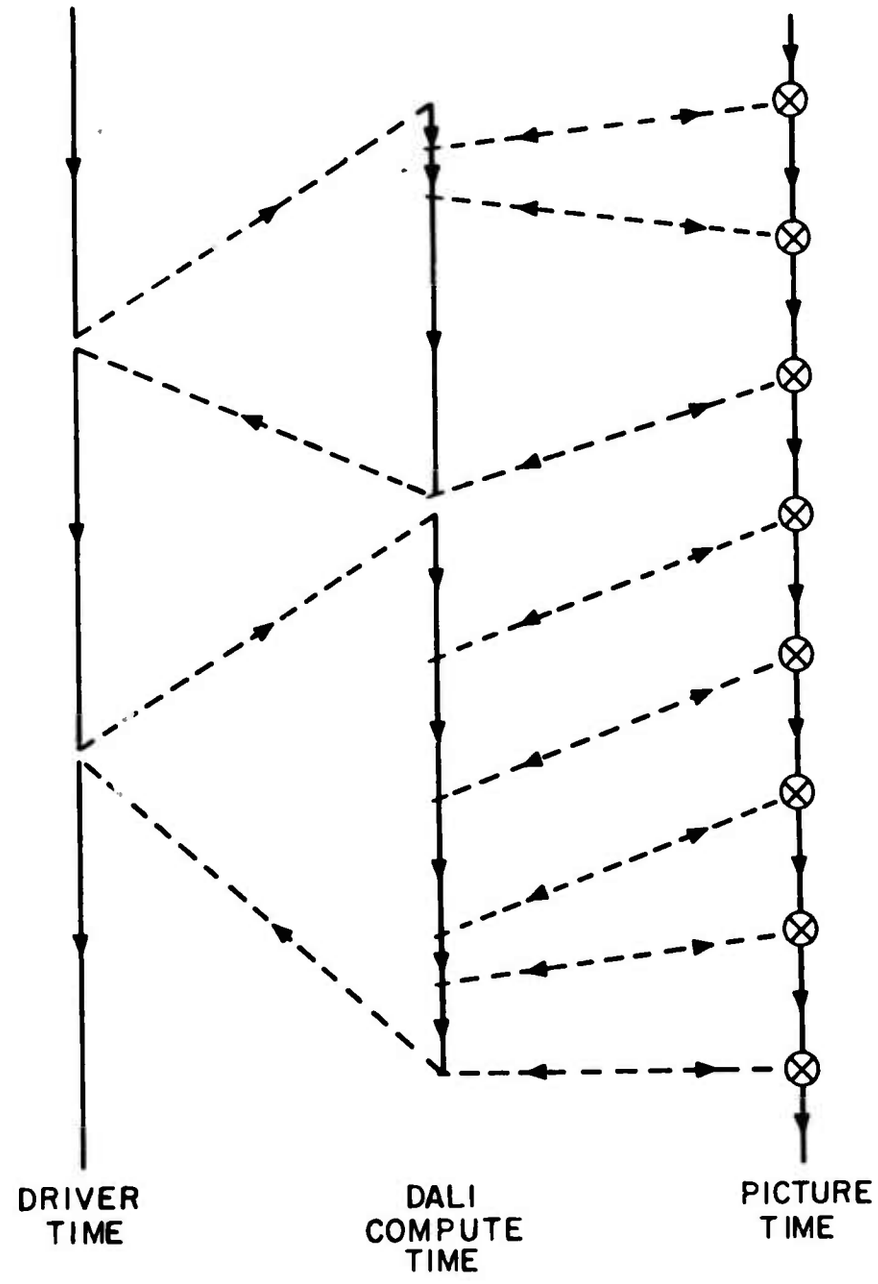


FIGURE 2 – 4 S – DALI OPERATION

changes to "watched" data. This provides a convenient method of interpolating values and creating the visual effect of smooth motion. In addition, the ability to create such future change sequences provides a mechanism for factoring the (picture-)temporal dynamics of pictures from time-invariant relationships such as connectivity, visual complexity as a function of scale, etc.: time-invariant operations can be done with M-DALI operations performed in response to value changes which are created and scheduled for future picture times by completely separate S-DALI operations. These M-DALI operations, performed during step 2.1, will of course refer to their eternal "now", i.e., the current picture time, and hence will affect the picture at the immediately succeeding step 2.2. This utilization of both M- and S-DALI operations in a single picture is illustrated in the example which follows.

The previous bar graph example will now be modified to interpolate the height of the bar, effectively moving it smoothly in picture time from one height to the next. Referring back to Fig. 2-2b, a straightforward way to accomplish this is by inserting an "interpolator" picture module between the driving process' value and the picture module T. This is module I in Fig. 2-5, which shows the modified picture definition. When I gets control due to a change in the driving process' value, it will apply a sequence of future changes to a value V, interpolating from V's current value to the value provided by the driving process. Module T now watches V, and so initiates the generation of many marginally different pictures. The total sequence of operation goes like this:

(1) The driving process changes its internal value from 3.0 to 4.5.

(2) I gets control during S-DALI step 2.1. Using V's initial value of 3.0, I applies a sequence of future changes to V, such as (3.1, 3.2, . . ., 4.5), choosing appropriate picture time intervals between the steps.

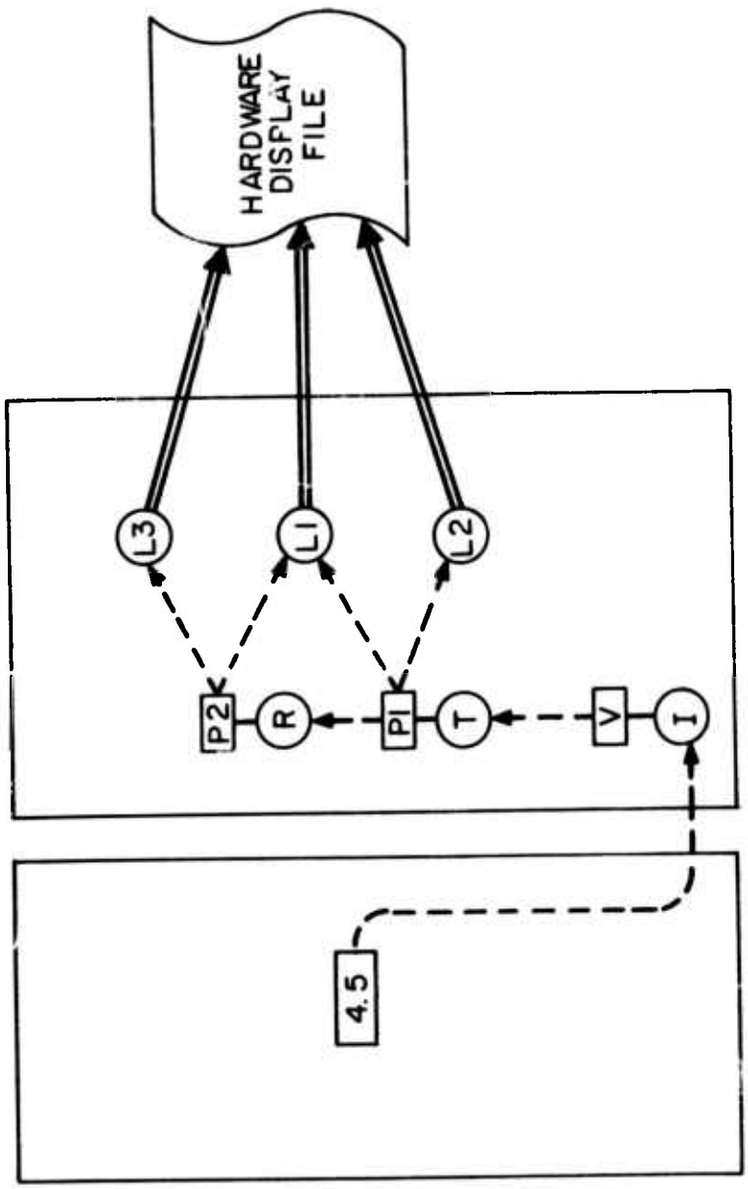


FIGURE 2-5 "SMOOTH" PICTURE CHANGE

(3) S-DALI step 2.1 terminates, and since I made no changes to values for the current picture time, S-DALI step 2.2 is null.

(4) S-DALI step 2.3 finds the change to V to 3.1, advances picture time appropriately, and cycles back to S-DALI step 2.1 to perform the change.

(5) In S-DALI step 2.1, V is changed to 3.1, then T runs changing P1, then R runs changing P2, then L1, L2 and L3 run.

(6) The picture changes scheduled for "now" by L1, L2, and L3 are performed as S-DALI step 2.2.

(7) S-DALI step 2.3 finds the future change of V to 3.2, advances picture time appropriately, and cycles back to S-DALI step 2.1 to perform the change.

.
.
.

(49) Having found the change of V to 4.5 at (46), S-DALI step 2.3 finds no further outstanding future changes and so proceeds to S-DALI step 3.

(50) Control returns to the driving process.

The process described above is one way to smooth out the bar's motion. There is at least one other: In S-DALI, it is possible to create a module replacing T which watches not for immediate changes in V but instead for the application of an entire sequence of future changes to V. This T′ module can then apply an appropriate sequence of future changes to P1, an action which can be detected by an R′ which can then apply an appropriate sequence to P2. Appropriate modules L1′, L2′, and L3′ can also be constructed to watch for P1 and P2 sequences and produce sequences of display file changes. With this method, the only repetitive work that need be done is the final output of the display file sequences.

In the chapters which follow, M-DALI will occupy most of our

attention. This is not because S-DALI is intrinsically less important or less interesting, but because many issues of interest are common to both S- and M-DALI and they are best discussed without the added complication of picture time.

2.4 Desiderata

This section discusses background issues which, while of minor inherent importance to the interesting issues of DALI's semantics, are necessary to an understanding of the exposition pursued in later chapters.

DALI is defined as a language extension. This has been done for two reasons: First, it allows the description to concentrate on those items which are unique to DALI and ignore irrelevant but necessary details such as the syntax of addition. Second, there is simply no reason to devise yet another language to express actions which are perfectly well expressed by existing languages, e.g., iteration and assignment.

The language of which DALI is an extension will be referred to as the base language. Essentially any programming language which is usable in practice can be a DALI base language; all that is required is that the base language (1) be arithmetically complete, i.e., be a Turing machine, and (2) contain some mechanism for defining and applying valued procedures or functions to arguments. No assumptions are made concerning the base language's control or environment structures.

The fact that many languages can be a DALI base language does not mean that creating a DALI extension is simple or even equally difficult for all base languages. Major additions to the base language's compiler or interpreter will be necessary, as will the construction of a fair

body of run-time support routines. For example, a class of assignment and identifier evaluation is defined by DALI, and run-time management of "heap" storage is needed.

Furthermore, not all DALI extensions will be equally powerful. For instance, DALI programs involving unpredictable storage demands will be significantly more difficult to write in a FORTRAN-DALI than in a LISP-DALI.

As a consequence of the fact that DALI is defined as a language extension, certain elements of DALI are undefined. In particular its syntax is undefined, as are various common elements of its semantics, e.g., whether or not it has a GOTO. While in the abstract this causes little difficulty, it creates certain expository problems, since examples become singularly difficult to present in the absence of a well-defined syntax. For this reason, in the chapters which follow the description will be couched in terms of a single sample base language, namely LISP. LISP was chosen for these reasons:

(1) It is fairly widely known.

(2) Its syntax is so simple that syntactic issues can effectively be ignored.

(3) Since the only operation directly performed by LISP programs is functional application, the semantics of an extension can be defined with relative ease.

The reader unfamiliar with LISP may consult [Weis1] or [McC1].

The principle semantic oddity which this choice forces upon DALI is that there are no unvalued procedures, only functions; however, the cases where a function returns no relevant value should be clear.

The fact that LISP uses retentive storage management -- a "garbage collector" -- is utilized in the "user code" of some of the examples; however, DALI constructs which are not in the base language not only do not require the use of a garbage collector, but also cannot utilize it. This is further discussed in section 4.3 and Appendix 3.

Of course, the syntax of LISP can be annoying. This is partially

alleviated here by the use of "prefix macro-characters" for certain much-used functions. Prefix macro-characters are used in MACLISP, the MIT Artificial Intelligence Laboratory's LISP [Moo1]. They are single characters which function as single-argument textual macros, transforming themselves and the single following LISP S-expression into a function call with a single argument. For example, one such macro-character used in DALI is comma (,), so that:

,expression

expands to

(OVAL expression) .

A character used as a macro-character is not usable as an identifier (ATOM) constituent.

To underline the fact that DALI should not be considered intimately bound to LISP, one example of DALI code is re-coded using a different base language: EULER. This example appears in section 3.12.

Independent of DALI being a language extension, however, it is in a real sense incomplete as presented here. Many features needed to round out DALI to its full extent have been left out; there are many further additions which could be incorporated, ranging from programmer convenience measures to additional constructs needed to make potential capabilities actual. For example, the problem of making pictures change according to the contents of arrays is nowhere treated, and neither is the equivalent problem concerning LISP lists. This has been done from a desire to address only those issues which are considered basic; hopefully, additional features can be readily conceived in terms of what has described. The reader's indulgence is begged if something of importance has been inadvertently left out.

There are three minor issues which must still be considered:

The first minor issue concerns terminology. Throughout this document, the LISP term "ATOM" and the term "identifier" will be used as

synonyms. This is not precisely correct usage, since, for example, a number is also an ATOM in LISP. When anything other than "identifier" is meant, some term other than "ATOM" will be used, e.g., "number".

The second minor issue concerns linguistic meta-syntax. The method used to describe the use of new functions is to give a sample application, such as

(POS x y)

and to describe in the text the function's arguments, value, and side effects as relevant.

Within a sample application, lower case items, such as x and y above, are syntactic meta-variables. Meta-variables beginning and ending with a dash (-) are used to represent multiple objects; otherwise each one represents a single object. Thus in the sample application

(ONS cndtn (-outs-) -body-)

cndtn is a single object, but -body- is some number of objects, and (-outs-) is a list of objects. Any limitations on the number of objects represented by a "dashed" meta-variable will be specified in the text.

The third minor issue concerns coordinate arithmetic. For convenience in expressing examples which use coordinate arithmetic, a very simple prior extension of LISP will be assumed. This extension, whose description follows, consists of the addition of a position data type and some associated functions.

Positions are created by the function POS. An application of the form

(POS x y)

where x and y are numbers, creates and returns a position which refers to the location (x,y) in some cartesian coordinate system.

Positions are examined by the functions X and Y. If p represents (POS x y), then:

(X p) returns x, and

(Y p) returns y.

The arithmetic functions for addition, subtraction, multiplication

and division (+, -, *, and /) are extended to operate on positions. If p is (POS x y), and p1 is (POS x1 y1), and n is a number, then:

(+ p p1) returns (POS (+ x x1) (+ y y1))

(- p p1) returns (POS (- x x1) (- y y1))

(* p n) returns (POS (* x n) (* y n))

(/ p n) returns (POS (/ x n) (/ y n)).

No operations on positions other than the above are defined.

The fact that positions are here defined to refer to a 2-dimensional space is not significant. DALI is concerned with control rather than calculation, and the dimensionality of the images it controls is of secondary importance.

Chapter 3
M-DALI: Basic Issues in Discontinuous Change

This chapter discusses the basic concepts behind M-DALI, a subset of DALI which deals only with discontinuous -- i.e., instantaneous or temporally Monadic -- change to pictures. M-DALI is the foundation of DALI, and the concepts and objects presented here are M-DALI's base.

Appendices 1 and 2 provide alphabetized lists and short description of all defined objects and procedures.

3.1 The Basic Objects: Outputs, Daemons, Picture Functions, and Picture Modules

As an extension to a base language, here LISP, DALI adds four primary types of objects: outputs, daemons, picture modules, and picture functions. The ways these four objects can be manipulated by the user, and the ways they interact in the DALI system, form the basis of M-DALI. This section briefly describes the purpose of each of the four, summarizes some of their characteristics, and illustrates how they are typically used with a very simple example.

The purpose of outputs is communication; they are used to hold data to be communicated between independent picture modules.

The purpose of daemons is processing; they are executable procedures.

The purpose of picture modules is hierarchical organization and provision for local storage; they are used as containers of objects, including other picture modules.

The purpose of picture functions is the creation of picture modules.

An output is an object containing a value. User-written procedures can access outputs' values and replace them with other values.

A daemon is a parameterless procedure executed in response to the occurrence of an event. One class of events that can cause daemons to respond is a change to the value of one or more outputs. By responding to output value changes, accessing output values, and changing output values, daemons propagate change in the manner mentioned in section 2.2.

A picture module is an organizational unit containing outputs, daemons, other picture modules, and a local environment providing data storage for daemons.

A picture function is a function, i.e., a valued procedure; it is applied to arguments and returns a value. Application of a picture function creates, in lieu of a procedural activation, a picture module which stays in existence until destroyed by the action of a user program (daemon). The value returned by a picture function is always the newly created picture module.

A simple example will now be given to illustrate how the four primary objects are typically used together.

The picture function RELP, which appears below, does not create any visible objects. Its purpose is to capture the notion of "relative position" in this sense: within the module created by applying RELP, a daemon is created which maintains the value of an output named SUM as the sum of the values of two other outputs. This SUM output can then, for example, be used as the endpoint of a line, assuming that the values added to obtain SUM's value are positions.

RELP can be defined as follows:

```
(DEFPIC RELP (P1 P2 "OUT" SUM)
   (ONS (VAL P1 P2) (SUM)
        (OUCH SUM (+ ,P1 ,P2)) ))
```

DEFPIC "declares" RELP to be a picture function taking two arguments: P1 and P2; these must be outputs. Applying RELP creates a picture module whose local environment holds P1, P2, and SUM. The indicator "OUT" means that SUM is a local temporary variable initialized to contain a null-valued output. The body of RELP, which is evaluated (executed) when RELP is applied, contains a single application of the function ONS; this application creates a daemon within the created module. This daemon runs whenever the value of P1 or the value of P2 changes; this is indicated by (VAL P1 P2). When run, this daemon performs a single functional application: it applies the function OUCH (for OUtput CHange) to change the value of SUM to the sum of the values of P1 and P2.

The output which is SUM's value can be obtained for use in other modules by means of the function OUT. Evaluating

(OUT (RELP A B))

first applies RELP to A and B, which are both outputs; this application returns the new module containing SUM. Then OUT is applied to the module and returns the SUM output. Since such applications of OUT are very common, a prefixed exclamation point (!) is used as syntactic sugar for such an application.

If P is an output which has a position as a value, and the outputs D1 and D2 contain positions used as vector displacements, the following code fragment creates a chain of relative displacements of length two:

(RELP !(RELP P D1) D2)

If only D2's value is changed, then only the daemon in the outer module is run; if D1, P, or both change value, the daemon in the inner module runs and then the daemon in the outer module runs.

Finally, assume LINE is a picture function creating a module which maintains a visible line connecting two endpoints defined by output values; when the endpoint values change, a daemon in this module changes the position of the visible line appropriately. Then

(LINE P !(RELP P D))

creates a "relative line" from P to P+D which will move appropriately if P's value, D's value, or both changes.

An LINE picture function with the properties assumed above will be defined in section 3.7.

3.2 The Operation of M-DALI

M-DALI's operation is simply explained in terms of daemon execution.

There is a single, global daemon queue which holds daemons to be executed. A daemon is placed on the queue as soon as its condition -- which describes the event the daemon responds to -- is satisfied. The order of daemons in the queue, and hence the order in which daemons will be run, is determined by a set of daemon scheduling rules which will be discussed in Section 3.8.

M-DALI's total operation consists of repeating the sequence below. The repetition begins when the driving process starts execution, and terminates when the driving process terminates.

(1) The driving process executes normally until some daemon's condition is satisfied, e.g., the driving process changes the value of an output watched by a daemon.

(2) The daemon(s) whose condition(s) have been satisfied are placed on the daemon queue, in an order determined by the scheduling rules, and control leaves the driving process.

(3) Until the daemon queue is empty, the daemon at the head of the queue is removed from the queue and executed. Any daemon whose condition is satisfied during such execution is queued; the currently running daemon is not interrupted.

(4) Control returns to the driving process, and step (1) above is re-entered.

Sections 3.8 and 3.9 contain further discussion of M-DALI's operation. The above, however, is all that M-DALI actually does on a

global level; everything else is determined by purely local interactions between outputs, daemons, picture modules, and picture functions.

3.3 Outputs and Daemons

An output contains a value, which may be any DALI or base language object. Outputs function with daemons as a mechanism for communication.

Functions concerned with outputs are:

create output:	(OUTPUT initial-value)
obtain value:	(OVAL output) or ,output
change value:	(OUCH output new-value)
destroy output:	(DELETE output)

OUTPUT returns the new output, OVAL (Output VALue) returns the value, OUCH (OUtput value CHange) returns the new value, and DELETE returns NIL. A prefixed comma (,) is syntactic sugar for an application of OVAL.

DELETE applied to any DALI object, not just an output, destroys that object; deletion is a complex operation and will be further discussed in section 4.3.

A daemon is a user-defined parameterless procedure which is executed in response to the occurrence of some event. The event a daemon responds to is described by its condition. Only certain classes of events can cause a daemon to respond; they will be introduced as they become relevant.

Among the events to which a daemon can respond, a particularly important class is a change to the value of one or more outputs. Since a daemon can access output values with OVAL and can use OUCH to change the value of an output, this class of events provides for the propagation of change as mentioned in section 2.2.

If a daemon's condition indicates that it is to be run when a particular output's value changes, that output is said to be watched by that daemon. An output whose value can be changed by the direct action of a daemon is said to be specified by that daemon. Many daemons can watch a single output, but only one daemon may specify, and hence change the value of, a given output. A daemon may specify any number of outputs.

To facilitate daemon queueing, each output contains (a pointer to) a list of the daemons watching it. Each output also contains its specifier to facilitate enforcing the "one specifier" rule. Both of these elements are null for a newly created output.

A daemon can be created by the function ONC (ON Condition) as in

(ONC cndtn (-s-outputs-) -body-) .

This returns the newly created daemon as a value. (-s-outputs-) is a list of the specified outputs; -body- is the executable body, and contains one or more statements (S-expressions); cndtn is the daemon's condition.

The function ONS (ON condition and at Startup) takes arguments and creates a daemon like ONC, but also runs the daemon's body once just before returning the completed daemon; this is extremely useful for initialization.

The condition

(VAL -outputs-)

will cause a daemon to be run whenever the VALue of any of the -outputs- is changed, i.e., when any of -outputs- are OUCHed. VAL is the condition used for the daemon in the preceding RELP example:

```
(ONS (VAL P1 P2) (SUM)
   (OUCH SUM (+ ,P1 ,P2))) .
```

The daemon created by this application watches the outputs assigned to P1 and P2, and specifies the output assigned to SUM.

It is often the case that a daemon should watch every output to which OVAL is applied in the daemon's body, and it is always the case

that the daemon should specify every output OUCHed in the body. Separate specification of watched and specified outputs is thus often redundant and somewhat error-prone. To alleviate this, two other daemon-creating functions exist: CONTIN and AS-NEEDED.

CONTIN, whose name was chosen to indicate CONTINuous and rhyme with begIN, is defined in terms of ONS:

(CONTIN -body-)

is equivalent to

(ONS (VAL -wouts-) (-souts-) -body-)

where -wouts- are all the outputs whose OVAL is explicitly referenced in -body-, and -souts- are all the outputs explicitly OUCHed in -body-. For example, the RELP daemon above could have been written as:

(CONTIN (OUCH SUM (+ ,P1 ,P2))) .

AS-NEEDED is similarly defined in terms of ONC.

It should be noted that CONTIN and AS-NEEDED work properly only if the arguments to OVAL and OUCH are simple identifiers which are assigned at daemon creation time to the outputs which the daemon will always reference. This is commonly the case. CONTIN and AS-NEEDED wil be used in preference to ONS and ONC whenever they are applicable.

The use of OUCH to cause daemons to run invokes a question: Does applying OUCH to an output always cause the daemons watching that output to be run, or do the daemons run only if the output's new value is not "equal" to its old value? The latter more precisely captures the notion of a daemon running in response to a value change, and so will be used. However, it involves the perennially thorny question of defining "equality" in the presence of data structures. This question is approached here on a pragmatic basis. The following defines OUCH's action:

Given an output outp whose current value is curv, the execution of (OUCH outp newv) has no effect if any of the following are true:

(1) curv and newv are both integers or both reals, and curv is

algebraically equal to newv within the precision being used.

(2) curv and newv are both positions which are elementwise equal according to (1) above.

(3) curv and newv are equal according to the simplest equality test available in the base language.

Otherwise, the value of outp is changed to newv and any daemons watching outp will be run according to the scheduling rules.

In the majority of languages, test (1) above will be subsumed under test (3). Test (1) is explicitly stated here both to make sure it is included and to facilitate the definition of test (2), which will normally not be included under test (3). The "simplest equality test available" in LISP is EQ [McC1], which, depending on the implementation, may or may not subsume test (1); but EQ does not include test (2).

When the phrase "an output is OUCHed" is used, it will be assumed that the output's value is changed and daemons are run.

To ease later discussion by eliminating a number of special cases, three conventions will be adopted:

(1) Normal procedures and functions, but not picture functions, will be considered "part of" the daemon which invokes them.

(2) The bodies of picture functions will be considered daemons in their own right.

(3) The driving program, the program running in the driving process, will be considered a daemon.

The latter two "daemons" -- picture function bodies and the driving program -- are anomalous with respect to how they are scheduled for execution. However, they are sufficiently similar to "normal daemons" in other ways that the above conventions produce fewer special cases than considering these "daemons" to be a different type of object. In particular, it can now be said that all user-specified processing done in DALI is performed by some daemon.

3.4 The Acyclic Data Web

The relations of watching and specifying between daemons and outputs can be diagrammed as shown in Fig. 3-1. This figure shows the watching/specifying relations assumed in the bar graph example of section 2.3. The conventions used in this and later similar diagrams are: daemons are large empty circles; outputs are small empty circles, connected to their specifier daemon with a short solid line; and a dashed arrow leads from an output to every daemon watching that output. The daemons and outputs of Fig. 3-1 are labelled to correspond with the objects in the less formal, and less correct, diagram of Fig. 2-2.

The relationships diagrammed in Fig. 3-1 describe the functional dependence of daemons upon other daemons, and are important in DALI. The collection of such relationships will be called the <u>data web</u> of a picture definition, or picture. While the data web can be completely arbitrary, it will initially be forced to contain no cycles, i.e., daemons directly or indirectly dependent on themselves. This will be enforced by two conditions on ONS and ONC which thus also apply to CONTIN and AS-NEEDED:

(1) A daemon may not be created specifying an output already specified by another daemon.

(2) A daemon may not be created watching an output not already specified by some daemon.

In the absence of primitives for making structural changes to the data web, these conditions prevent the creation of functional circularity via daemons and outputs, i.e., data web cycles; the situation is similar to a LISP system containing CONS, CAR, and CDR, but neither RPLACA nor RPLACD. Data web structural change primitives are introduced in section 4.4, and cycles are considered in Chapter 5. ONC and ONS, however, will <u>always</u> obey the above conditions; another daemon-creating primitive is introduced in section 5.4 for the purpose of creating data web cycles.

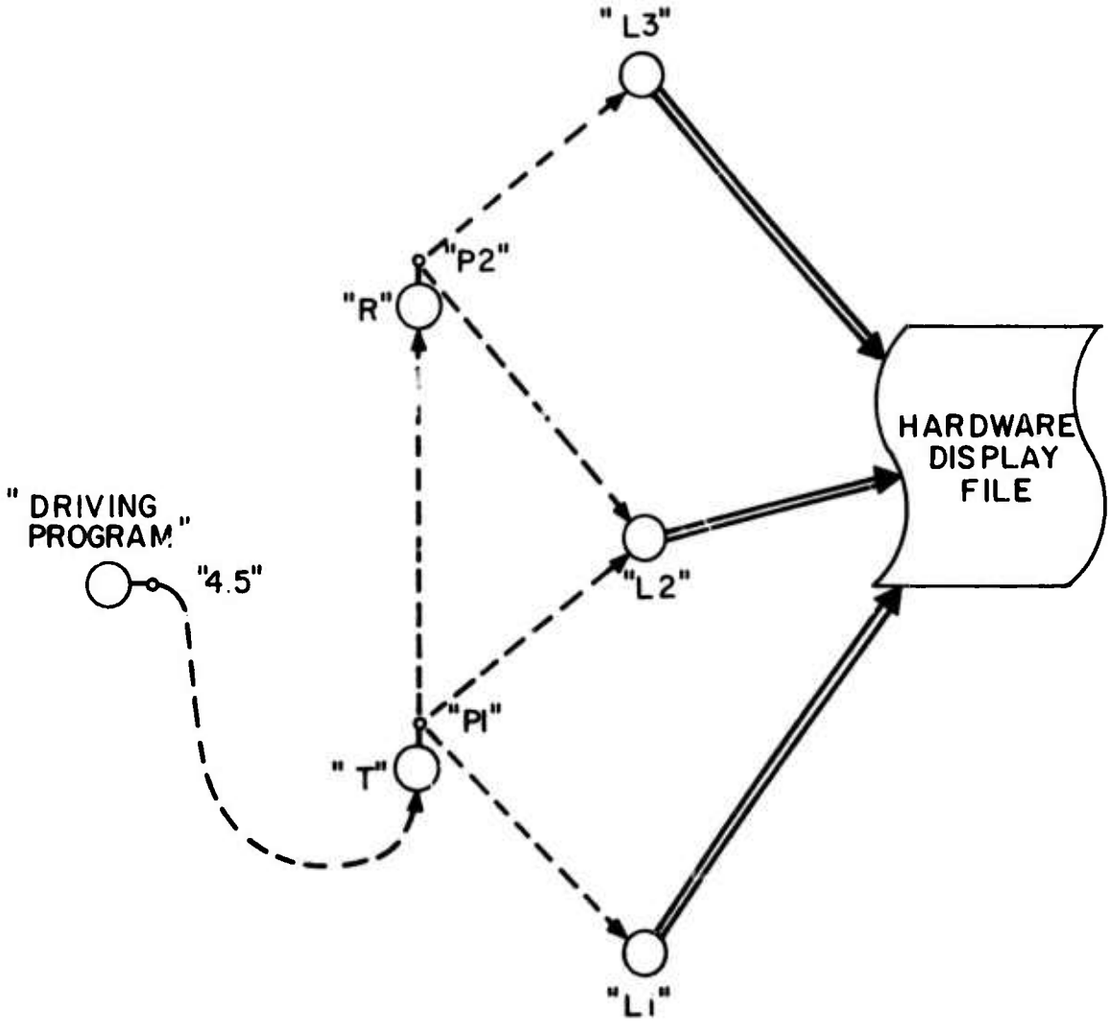


FIGURE 3-1 DATA WEB FOR THE BAR

The acyclic condition labelled (2), above, can be inconvenient when "constant" outputs, whose values are never to change, are used. To alleviate this, the function NULLSPEC, for NULL SPECifier, exists:

(NULLSPEC outp)

where outp is an output, causes outp's specifier to become a "null" daemon which watches no outputs and never runs. This allows daemons to be created watching outp, even though outp has no real specifier. This is often useful for uniformity, since functions and picture functions designed to use outputs will not work correctly if passed non-output values.

In addition, an output with no specifier, not even the "null" daemon mentioned above, may be OUCHed by anybody. This does no harm, since no daemon can watch such an output; and it is occasionally useful in initialization.

3.5 Picture Modules, the Containment Tree, and the Picture Structure

A picture module, or just module, is an organizational unit containing outputs and daemons. A picture module is said to own the daemons and outputs it contains. Every daemon or output has one and only one owner picture module. The owner of a daemon becomes the owner of every daemon or output created by that daemon.

Fig. 3-2 extends Fig. 3-1 to show ownership of daemons by modules. Ownership of outputs will not be diagrammed, simply because doing so would increase the complexity of the diagrams intolerably. A further convention is introduced in Fig. 3-2: modules are cross-hatched circles, connected to the daemons they own by dotted lines. Although it does not occur in Fig. 3-2, a module can own several or no daemons.

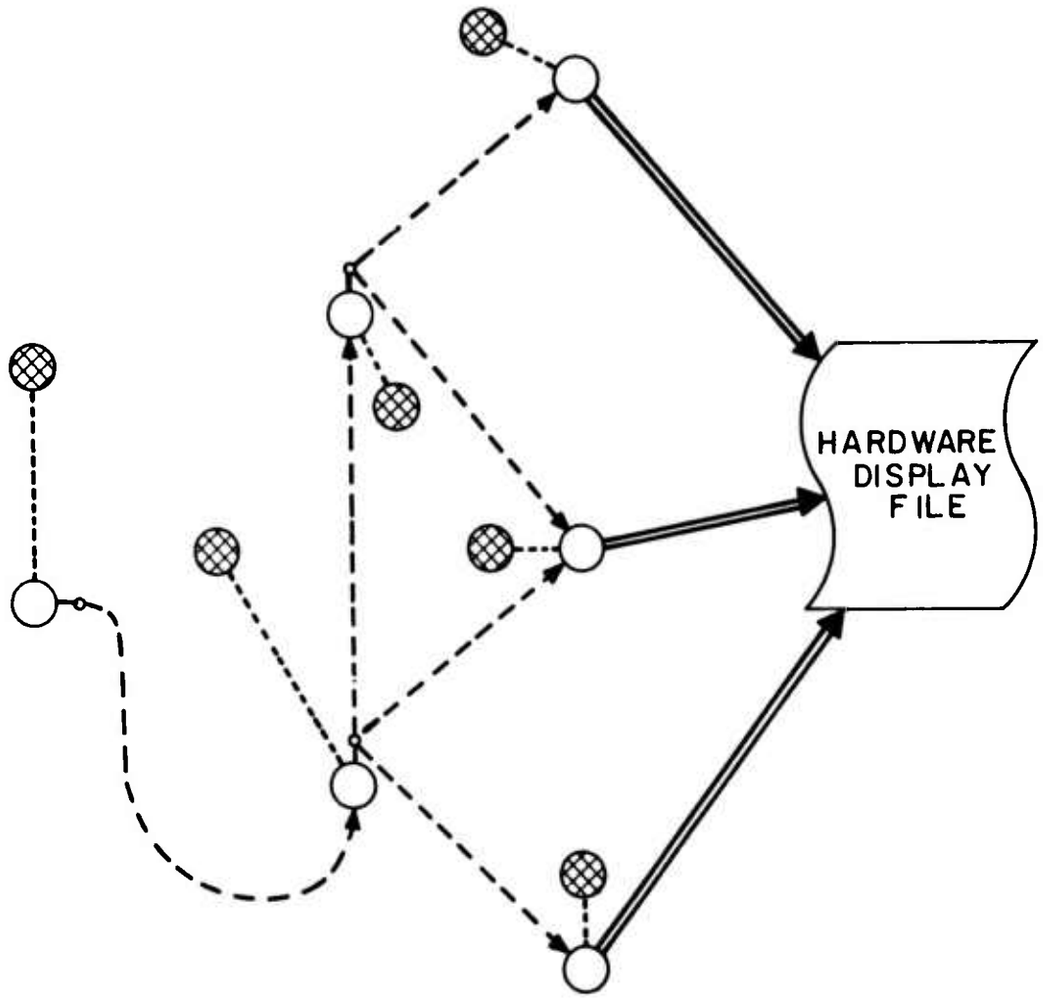


FIGURE 3-2 DAEMON OWNERSHIP IN THE BAR

Every module also contains a local environment, which is a mapping from identifiers to values. The local environment of a module is unique to that module, and does not change in size or structure during execution: both the number of identifiers in its domain ("bindings") and its organization are fixed at the time the module is created.

The environment used for daemon execution contains more than just the local environment; this will be discussed in section 3.11.

Local environments provide daemons with private storage which is retained across separate daemon executions. All of the daemons owned by a module have access to all of their owner's local environment, and may obtain and change the value of any identifier therein. Normal identifier evaluation obtains the value, and normal assignment (SET and SETQ) changes the value.

No daemon not owned by a module can change the value of any identifier in that module's local environment.

A number of the identifiers in the local environment may be designated output identifiers. Daemons not owned by a module can obtain the values of that module's output identifiers by use of the function OUT:

(OUT picture-module integer)

returns the value of the integerth output identifier of picture-module. The integer is optional and defaults to 1; if given, it must be greater than 0. The manner in which the implied ordering of output identifiers is accomplished is discussed in section 3.6, Picture Functions.

Since OUT is often used with the default argument of 1, an abbreviation for such an application will be used: a prefixed exclamation point (!). Thus the following all represent the same thing:

(OUT something 1) (OUT something) !something

Neither a module nor its daemons has any control over what daemons can obtain output identifiers' values by using OUT.

Output identifiers normally have values which are outputs, although

this is not always the case. An output which is the value of an output identifier will occasionally be referred to as an "output of a module".

In addition to daemons, outputs, and a local environment, a picture module can also contain other picture modules. If a picture module A contains a picture module B, A is called the father of B and B is called a son of A. The owner of a daemon creating a picture module becomes the new module's father.

Father-son relationships form a tree structure called the containment tree. The root node of the containment tree is called the root module when it is not being called the driving process.

Fig. 3-3 adds the containment tree to the "bar" example, using the last convention for such figures: solid arrows point from fathers to their sons. A picture module which contains no daemons has been added to make the structure shown in Fig. 3-3 a true substructure of that generated by the written examples which will follow in section 3.12.

The primary purpose of the containment tree is to propagate deletion: when a module is deleted, its sons are also deleted. This must be done if a module is to be considered a unit whose internal details of operation, including the creation of sons, are transparent to any use of the module, including its deletion. For this same reason, all the objects owned by a module, including daemons and outputs in addition to sons, are deleted when the module is deleted. Deletion is further discussed in section 4.3.

The union of the containment tree, the data web, and owner relations is called the picture structure of a picture definition or picture.

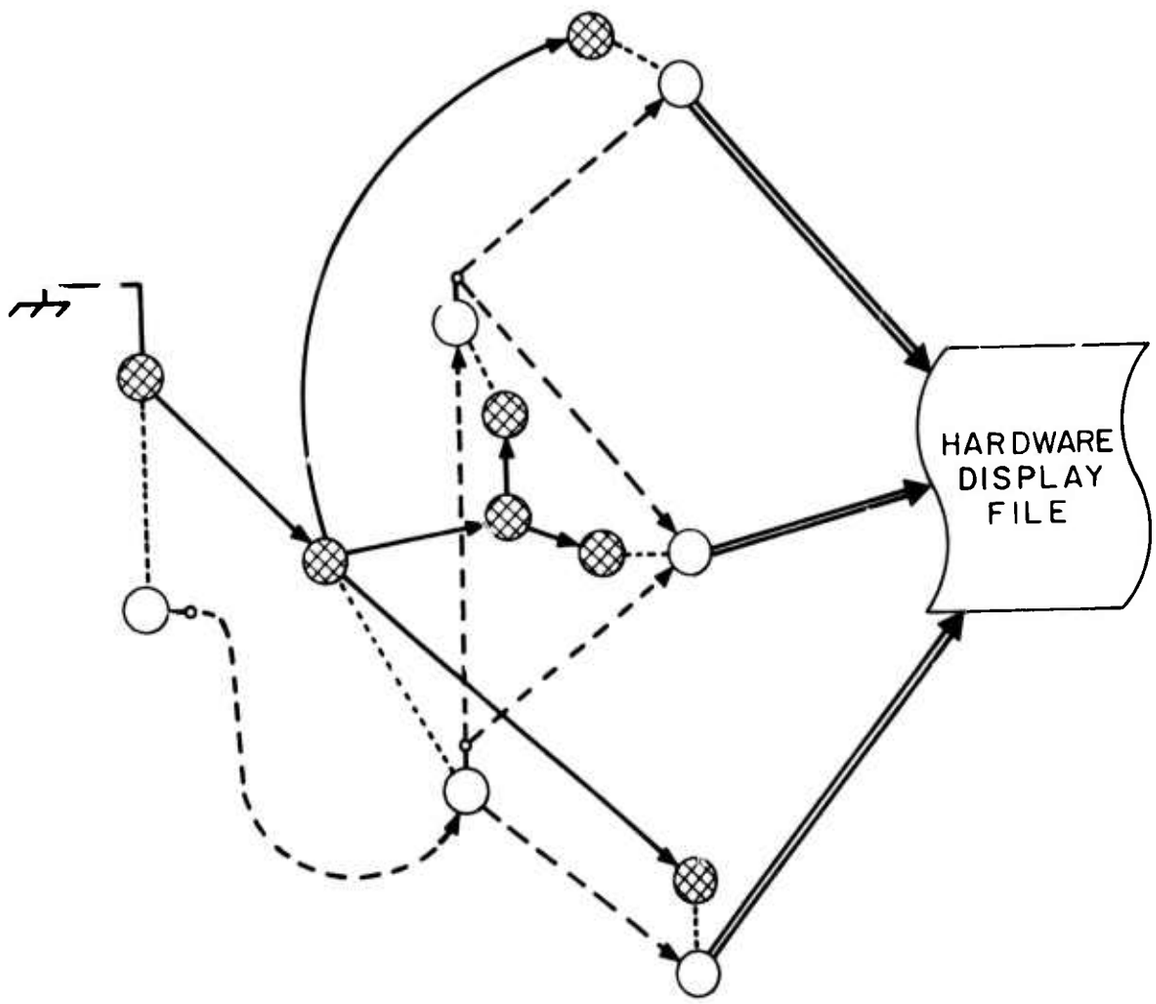


FIGURE 3-3 PICTURE STRUCTURE OF THE BAR

3.6 Picture Functions

A picture function is a user-defined function, i.e., a valued procedure, of a special type. Applying a picture function to zero or more arguments always creates a new picture module and returns that new picture module as the value of the application. The arguments applied to a picture function will occasionally be referred to as "inputs" of the picture module created.

The characteristics of picture functions are bound up in the process of applying them.

In applying a picture function, the first thing which happens is that the new picture module is created. The local environment of the new module is defined by the argument list ("declarations") of the picture function, and the applied arguments are assigned to identifiers in that local environment. Then the body of "code" in the picture function is executed as a daemon of the new module. Finally, the new module is returned as the value of the picture function.

A picture module is to a picture function as a procedural activation is to a procedure; or, in more classical computer graphics terms, a picture module is to a picture function as an instance is to a master.

A picture function has an argument list and a body. The argument list is described later in this section; the body is one or more executable statements (S-expressions). An unnamed picture function is represented as an application of PICTURE, analogous to LISP's LAMBDA:

(PICTURE (-argument-list-) -body-)

is a picture function whose argument list is (-argument-list-), and whose body is -body-.

The function DEFPIC, for DEFine PICture function, is used to "declare" an identifier to be a picture function. In the usable LISP

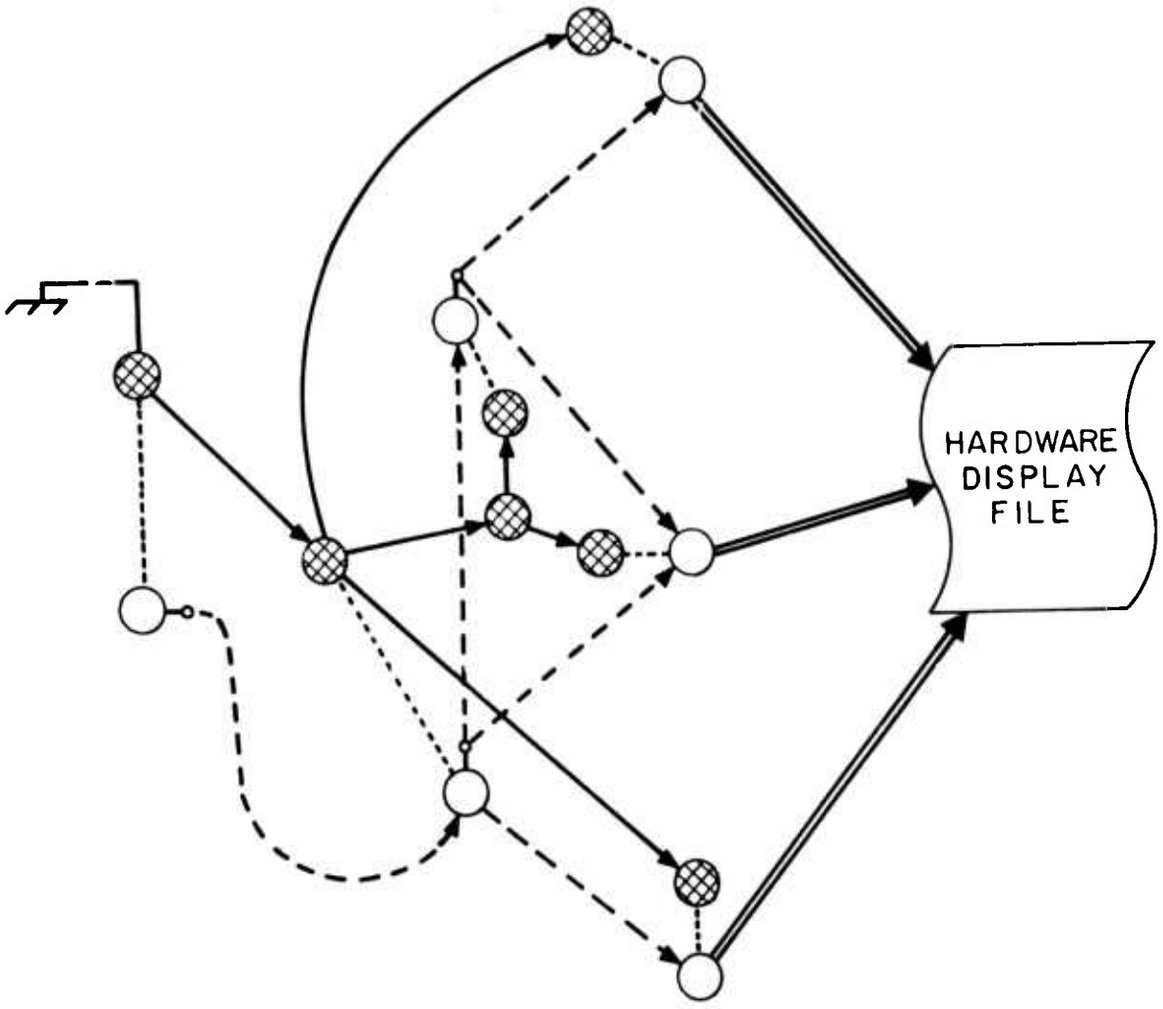


FIGURE 3-3 PICTURE STRUCTURE OF THE BAR

3.6 Picture Functions

A picture function is a user-defined function, i.e., a valued procedure, of a special type. Applying a picture function to zero or more arguments always creates a new picture module and returns that new picture module as the value of the application. The arguments applied to a picture function will occasionally be referred to as "inputs" of the picture module created.

The characteristics of picture functions are bound up in the process of applying them.

In applying a picture function, the first thing which happens is that the new picture module is created. The local environment of the new module is defined by the argument list ("declarations") of the picture function, and the applied arguments are assigned to identifiers in that local environment. Then the body of "code" in the picture function is executed as a daemon of the new module. Finally, the new module is returned as the value of the picture function.

A picture module is to a picture function as a procedural activation is to a procedure; or, in more classical computer graphics terms, a picture module is to a picture function as an instance is to a master.

A picture function has an argument list and a body. The argument list is described later in this section; the body is one or more executable statements (S-expressions). An unnamed picture function is represented as an application of PICTURE, analogous to LISP's LAMBDA:

(PICTURE (-argument-list-) -body-)

is a picture function whose argument list is (-argument-list-), and whose body is -body-.

The function DEFPIC, for DEFine PICture function, is used to "declare" an identifier to be a picture function. In the usable LISP

system assumed here, DEFPIC actually gives some ATOM a FUNCTION property which is a picture function. In "canonical" LISP, closer to pure lambda-calculus, less convenient mechanisms are used.

A DEFPIC application returns NIL, and looks like this:

(DEFPIC name (-argument-list-) -body-) .

name is an ATOM naming the picture function, -body- is the body of the picture function, and (-argument-list-) is the argument list.

Returning to our simple RELP picture function:

```
(DEFPIC RELP (P1 P2 "OUT" SUM)
   (CONTIN (OUCH SUM (+ ,P1 ,P2))))
```

declares RELP to be the picture function

```
(PICTURE (P1 P2 "OUT" SUM)
   (CONTIN (OUCH SUM (+ ,P1 ,P2))))
```

whose argument list is

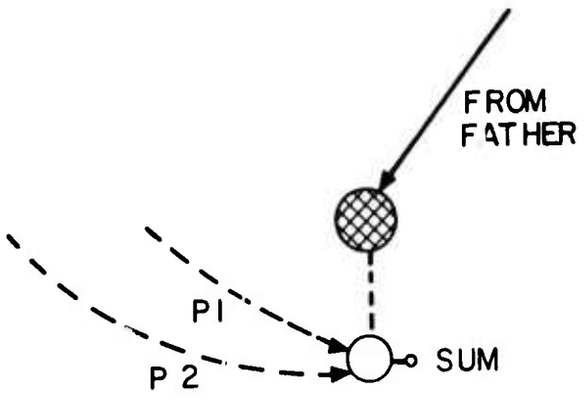
```
(P1 P2 "OUT" SUM)
```

and whose body is

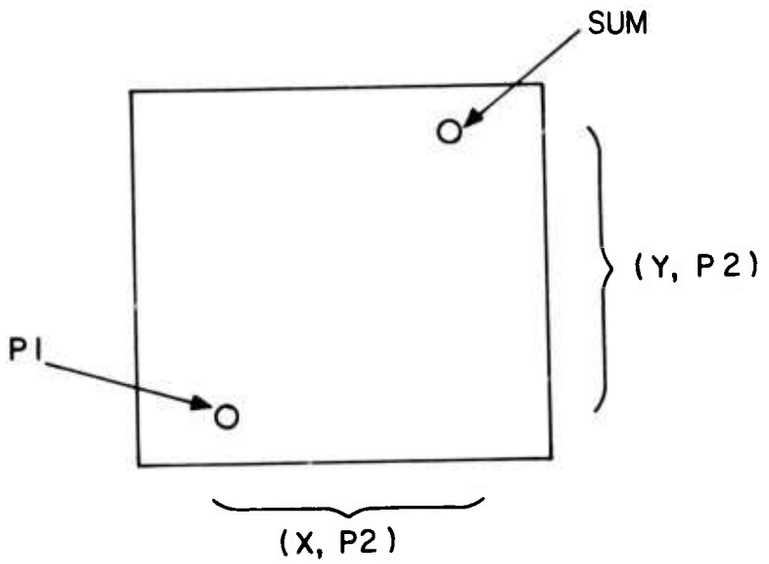
```
(CONTIN (OUCH SUM (+ ,P1 ,P2))) .
```

"OUT" in the argument list signals that SUM is an output identifier which, for convenience, is automatically initially assigned to (OUTPUT NIL). The body of RELP is a single statement which creates a daemon. Since CONTIN, defined in terms of ONS, is used, the body of the daemon is run once just after creating the daemon; thus SUM's value is conveniently initialized. RELP captures the notion of a "relative position" in that it provides an output -- SUM -- which is continuously maintained as the sum of two other position-valued outputs; one of these is the base position, the other is the vector displacement. Applying RELP creates the picture substructure shown in Fig. 3-4a; RELP's graphical effect is shown in Fig. 3-4b.

The syntax and semantics of picture function argument lists remains to be explained. The syntax used is based on the argument list syntax developed for the MUDDLE language [Pfi2] by C. Reeve of MIT Project MAC.



A



B

FIGURE 3-4 RELP

The argument list of a picture function is a list of ATOMs of any finite length divided into sections by the designators "EXTERNAL", "AUX", "AUXO", "OUT", and "OUTU". An example:

```
(A B C "AUX" D E "EXTERNAL" QWERT L ARGLE
 "AUXO" MNO P "OUTU" BARGLE "OUT" SUM DIF)
```

No ATOM or designator may occur more than once in an argument list, and the order in which the designators occur is not significant. All of the ATOMs in the argument list, and only those ATOMs, are identifiers in the local environment of the module created. The effect of the designators on the local environment is the principle issue here.

The term "ATOMs following XXX" will be used to mean "ATOMs to the right of XXX in the argument list, and to the left of either (1) the end of the argument list or (2) the first designator to the right of XXX, whichever comes first." For example, D and E are ATOMs following "AUX" in the above example, and ATOMs following the start of that argument list are A, B, and C.

The ATOMs following the start of the argument list are initially assigned to the arguments applied to the picture function in the normal left-to-right order. The LISP equivalent of call by value is used, i.e., the expressions in the application are evaluated and the result obtained is actually assigned in the local environment. This is a function of the base language used to host a DALI extension, and will have some effect on how DALI appears to the user. These "argument" ATOMs are used for initial communication into the picture module.

The ATOMs following "AUX" are initially assigned to NIL, and the ATOMs following "AUXO" are initially assigned to (OUTPUT NIL). These ATOMs are used for purely local internal storage by the picture module. The mnemonic value of these two designators is "AUXiliary".

The ATOMs following "OUTU" are initially assigned to NIL, and the ATOMs following "OUT" are initially assigned to (OUTPUT NIL). These ATOMs are the output identifiers of the picture module, numbered starting with 1 in the order of their left to right occurrence in the argument list. They are used for communication out of the picture

module, since their values can be obtained by using the function OUT as described in the previous section. "OUTU" stands for OUTput Uninitialized.

The new outputs created by "AUXO", for AUXiliary Output, and "OUT", for OUTput, are owned by the new picture module. This output creation is a convenience measure only; the same effect could be achieved by assignments performed in the picture function body.

The ATOMs following "EXTERNAL" are "invisible arguments". Each such ATOM is initially assigned to the value of the identical ATOM in the local environment "closest" to the new module on the path from the new module through its father to the root module. The implied search is performed only once, namely when the environment is created. Each "EXTERNAL" ATOM has its own "binding slot" in the new local environment, distinct from that where its initial value was found. Thus, "EXTERNAL" ATOMs are not actually "free" or "fluid" variables, since assignment to them affects only the newly created local environment. They are primarily a mechanism for avoiding argument lists of unwieldy length and passing data across modules not interested in that data. If it is necessary to dynamically vary the data passed in this manner, the data can always be encapsulated in an output.

The five designators discussed above -- "EXTERNAL", "AUX", "AUXO", "OUT", and "OUTU" -- are, for simplicity, the only ones which will be defined and used in this document. Many other useful features could be added, epitomized by the facilities of MUDDLE [Pfi2]: optional arguments, an arbitrary number of arguments, user-specifiable value initialization, unevaluated arguments (LISP's equivalent of call by name), etc.

3.7 Examples: Coding the BAR

An implementation will now be given for the picture function LINE which was used in the relative position examples of section 3.1.

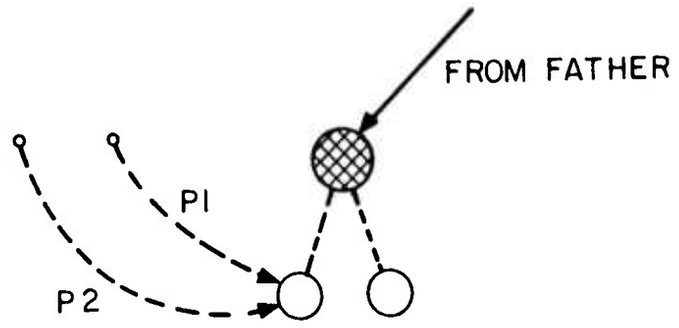
The characteristics desired of LINE modules are four: (1) they have two inputs, both position-valued outputs, which dictate the positions of the drawn line's endpoints; (2) when a LINE module is first created, an associated entry in the hardware display file is created by the picture function body and remembered in the local environment; (3) when either endpoint is changed, the entry in the display file is changed accordingly by a daemon; (4) when the module is deleted, a daemon deletes the entry in the display file. LINE can be coded as follows:

```
(DEFPIC LINE (P1 P2 "AUX" LINEID)
   (SETQ LINEID (MAKE-LINE-ENTRY ,P1 ,P2))
   (AS-NEEDED (CHANGE-LINE-ENTRY LINEID ,P1 ,P2))
   (ONC DELETE () (DESTROY-LINE-ENTRY LINEID))  )
```

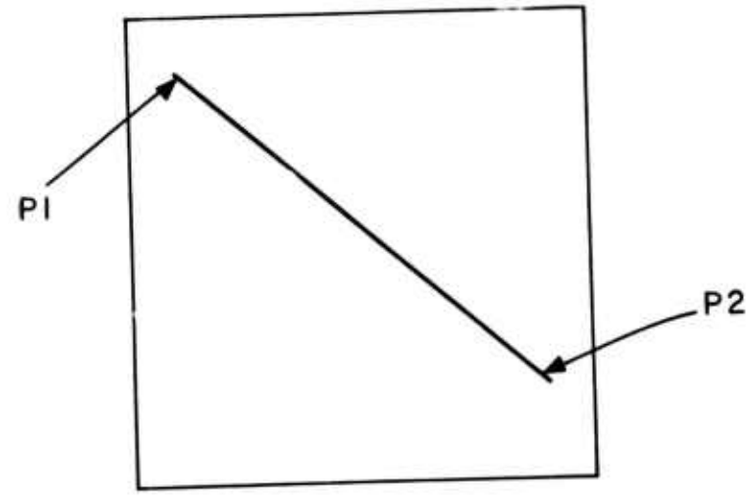
LINEID is used to record some object identifying the hardware display file entry created by MAKE-LINE-ENTRY. This object is passed to CHANGE-LINE-ENTRY, along with the new endpoint values, when either endpoint changes. LINEID's value is also used to indicate which entry to delete when the module is deleted. Neither daemon specifies any outputs; if the AS-NEEDED daemon were explicitly coded with ONC, it, like the second daemon, would have an empty specifier list. The second daemon utilizes the condition DELETE, which causes it to be run just before LINE is actually destroyed. This condition is further discussed in section 4.3.

The picture substructure created by an application of LINE is shown in Fig. 3-5a, and the graphical effect of a LINE is shown in Fig. 3-5b.

LINE and the previously defined RELP module can be used to create a "relative line" picture function. This is RELINE, below. RELP is repeated for convenience.



A



B

FIGURE 3-5 LINE

```
(DEFPIC RELINE (P1 DELTA "OUTU" P2)
   (SETQ P2 !(RELP P1 DELTA))
   (LINE P1 P2))

(DEFPIC RELP (P1 P2 "OUT" SUM)
   (CONTIN (OUCH SUM (+ ,P1 ,P2)))
```

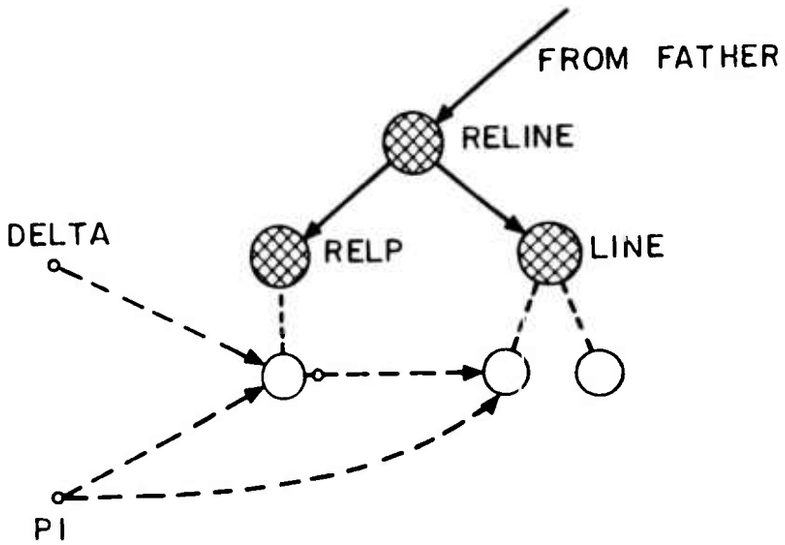
The inputs to RELINE, P1 and DELTA, are again position-valued outputs respectively specifying one endpoint and a vector displacement. Recall that a prefixed ! is syntactic sugar for an application of OUT. By assigning RELP's output to P2, RELINE has made that output available to its own callers; this will prove convenient later. RELP is <u>not</u> called with the <u>values</u> of P1 and DELTA; instead the outputs themselves are passed. Similarly, outputs are passed directly to LINE. Thus RELINE need contain no daemons, since its containment tree sons do all the work. RELINE itself serves only as a mechanism for treating this particular construct as a unit which is created, destroyed, and -- as far as its caller is concerned -- operates as a whole.

Fig. 3-6a shows the picture substructure created by an application of RELINE, and Fig. 3-6b shows the resultant picture.

In RELINE, the responsibility for forming a relative position and for drawing a line was effectively delegated to RELINE's sons. Such delegation of authority need not be total. Suppose, as a simple example, that a triangle is to be drawn but that one of its endpoints should be restricted to having only a certain range of "safe" values; if an endpoint goes out of the "safe" range, "drastic" action is to be taken. This can be done as shown in WATCHP3 below. A semicolon indicates that the remainder of the line is a comment.

```
(DEFPIC WATCHP3 (P1 P2 P3 "AUXO" FILTER)
;If P3 is safe, send it on via FILTER; else call DRASTIC.
   (CONTIN (COND ((SAFE? ,P3) (OUCH FILTER ,P3))
                 (T (DRASTIC)) ))
;Draw the triangle, using P1, P2, and FILTER.
   (LINE P1 P2)
   (LINE P2 FILTER)
   (LINE FILTER P1) )
```

WATCHP3 itself responds only when P3's value changes. It never needs to see changes to P1 and P2, since its LINE sons take care of them.



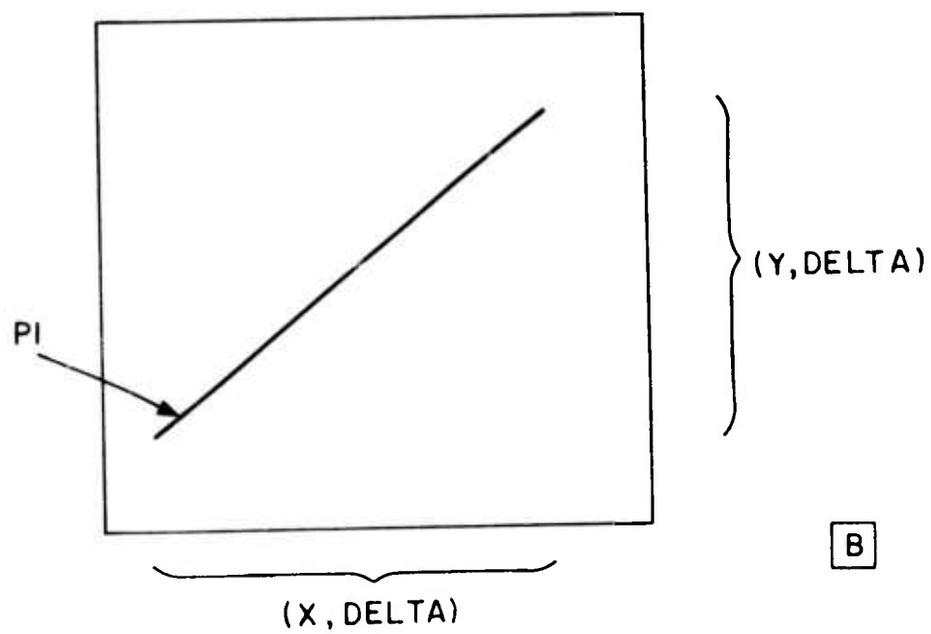


FIGURE 3-6 RELINE

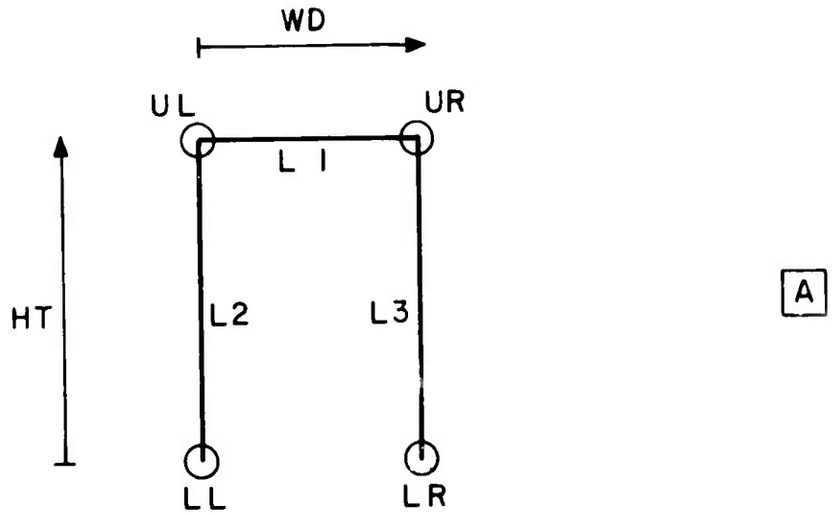
Now a picture function to draw a bar of the bar graph will be presented. This picture function, BAR, will draw left, top, and right sides of the bar. BAR takes four arguments, all outputs: (1) V, the numeric value to be shown; (2) SCL, the scale factor which, multiplied by V, gives the desired height; (3) LL, the position of the lower left corner of the bar; and (4) WIDTH, a number giving the desired width of the bar. A coding of BAR is:

```
(DEFPIC BAR (V SCL LL WIDTH "AUXO" HT WD)
   (CONTIN (OUCH HT (POS 0 (* ,V ,SCL))))
   (CONTIN (OUCH WD (POS ,WIDTH 0)))
   (LINE !(RELP LL WD) !(RELINE !(RELINE LL HT) WD)) )
```

The two daemons are straightforward. The first places in the HT output the real displayed height of the bar, computed from the given value V and the scale factor SCL. The second converts WIDTH into the vector displacement desired by RELP and RELINE, placing it in the WD output. The last expression is a minor exercise in embedding. The outermost picture function, LINE, draws the right side of the bar. The RELP generates the lower right corner. The innermost RELINE draws the left side of the bar and provides the upper left corner of the bar as an output. The outer RELINE uses this and the width to draw the top of the bar, and provides the upper right corner as an output so that the LINE module can draw the right side.

Fig. 3-7a shows the picture BAR draws. The lines drawn and the position outputs used are labelled, and graphical interpretations of WD and HT are given. Fig. 3-7b uses those labels to make part of the picture structure created by BAR intelligible: the daemons changing LINE display file entries are labelled as the corresponding lines, and outputs are labelled as named in BAR or as in Fig. 3-7a, and modules are labelled with the names of their picture functions. All daemons shown as ownerless are owned by BAR, and all modules shown as fatherless are sons of BAR.

More complex examples are given in section 3.12.



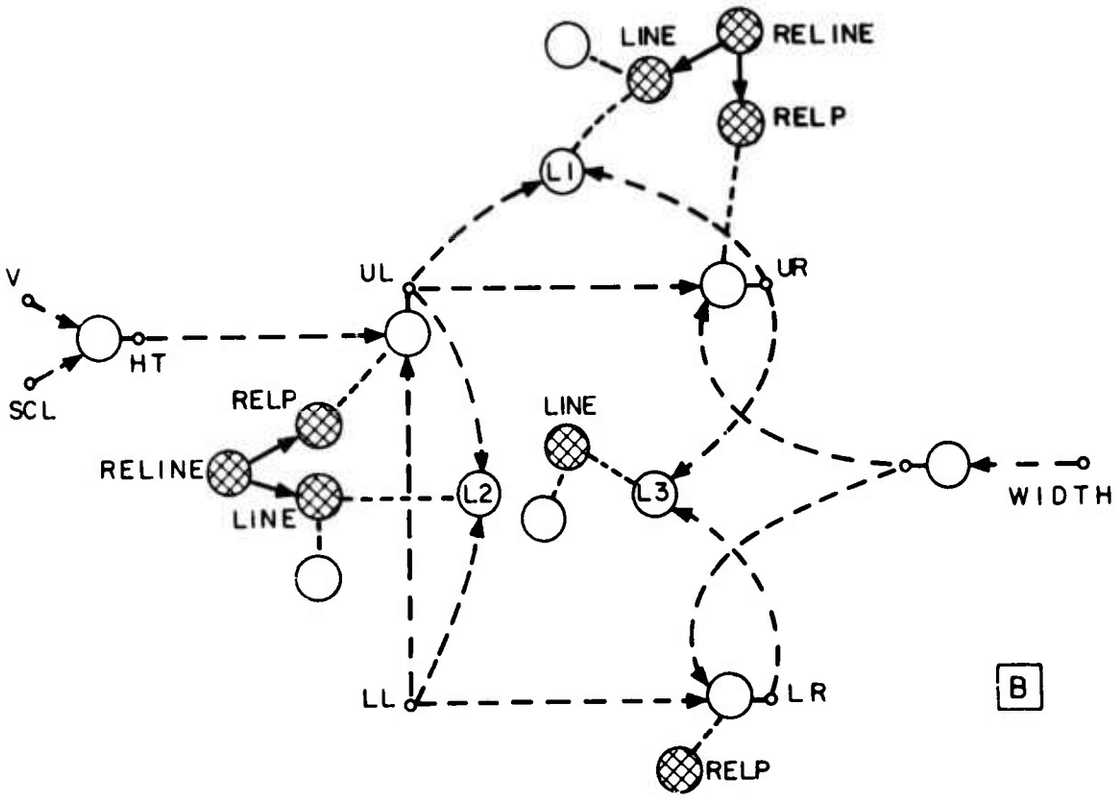


FIGURE 3-7 BAR

3.8 The Acyclic Daemon Scheduling Rules and Their Implementation

The daemon scheduling rules govern the running of daemons during DALI compute time. I.e., the scheduling rules define when and where daemons are placed on the daemon queue discussed in section 3.2. The scheduling rules presented here deal only with daemons which watch outputs. Daemons which have other conditions, e.g., DELETE, are scheduled as special case exceptions to the given rules; such exceptions will be explained when other conditions are covered in detail. The rules discussed also ignore deletion, in that they implicitly assume that a daemon whose watched outputs are OUCHed will always be run; obviously, such a daemon could be destroyed after it is queued but before it gets the chance to run.

The scheduling rules given in this section deal only with daemons in an acyclic data web; this was described in section 3.4 and will be more rigorously defined later in this section. This special case is important since it occurs in many situations of interest, can be handled simply, and relates to the general case of cyclic data webs discussed in Chapter 5.

Starting with some necessary definitions, the four acyclic daemon scheduling rules are presented, followed by discussion of their implementation. A later section, 3.10, deals with the related issue of interrupts from external devices.

The needed definitions are these:

> A daemon A is a web father of a daemon B, and B is a web son of A, if and only if A specifies an output watched by B.
>
> A daemon A is a web ancestor of a daemon B, and B is a web descendant of A, if and only if there exists a sequence of daemons D(0),D(1)...,D(n) such

that D(0)=A, D(n)=B, and for every i in the range 0<i<n+1, D(i-1) is a web son of D(i).

A data web is said to be acyclic if and only if it contains no daemon which is its own web ancestor.

The acyclic daemon scheduling rules defining the order of daemon execution in an acyclic data web are:

Rule 1: (selection) A daemon will be run if and only if one or more of its watched outputs has been OUCHed; once run, it does not run again until such an OUCH occurs again.

Rule 2: (noninterruption) Once a daemon D begins execution, no daemon web-ancestrally related to D may run until D terminates of its own accord.

Rule 3: (ancestors first) If daemons A and B are to be run, and A is a web ancestor of B, then A is run before B.

Rule 4: (closure) If two daemons are to be run and they are not related by web ancestry, they may run in any order.

Since the data web in acyclic, no two daemons can be web ancestors of each other; thus Rule 3 is deterministic and Rule 4 does cover all cases not covered by Rule 3.

Implementation of the acyclic daemon scheduling rules for a single processor system is relatively simple and efficient when the only data web changes arise from daemon creation and destruction. Modifications to the implementation given here will be discussed when other changes to the data web are introduced in section 4.4.

Each daemon is assigned an unchanging integer priority at its creation according to the following rules:

(1) The priority of the driving program is 0.

(2) The priority of a daemon is one greater than the largest of its web fathers' priorities.

The web fathers of a daemon can be easily found: they are the specifier element of its watched outputs.

Whenever an output is OUCHed, each daemon watching that output is immediately added to the daemon queue if it is not there already. Daemons are queued in order of increasing daemon priority; i.e., the daemon at the head of the queue has the lowest priority in the queue. The running of daemons, as described in section 3.2, then consists of removing the daemon at the head of the queue from the queue and running it, repeating this until the queue is emptied.

The fact that daemons are only queued in response to an OUCH satisfies the selection rule. The noninterruption rule is satisfied because daemons are always queued before they are run. Ordering the queue by increasing priority satisfies the ancestors first rule, since a daemon's priority must be greater than that of all its web ancestors.

The queueing process can be sped up somewhat in two ways: (1) Searching the data web for already queued daemons can be eliminated by including in each daemon a flag set when the daemon is queued and reset when it is removed from the queue for running. (2) All the daemons watching an output can be queued in a single sort/merge through the queue if the list of daemons watching each output is pre-sorted by increasing priority.

Two already mentioned types of "daemons" are exceptions to the acyclic daemons scheduling rules: picture function bodies and the driving program. Both are exceptions to Rule 2, noninterruption, and both are daemons only by convention.

Picture function bodies violate Rule 2 in that they are executed embedded in the execution of the daemon calling them. Delaying their execution until their calling daemon returns would result in severe initialization problems: the outputs created by the called picture function, which are definitely of interest to the caller, would not exist until after the caller terminated its execution.

The driving program clearly must violate Rule 2 to allow for the execution of any other daemon. Here again, the execution of other

daemons is strictly embedded as if it were an extended subroutine or procedure call. Performance of an OUCH in the driving program just causes daemons to be queued for execution as usual. Once the desired set of changes has been made, the driving program calls the parameterless procedure UPDATE-DISPLAY; this initiates the normal repetitive running of queued daemons until the daemon queue is exhausted, and then returns back to the driving program. UPDATE-DISPLAY has no effect if called from other thar the driving program.

3.9 Goals of the Acyclic Daemon Scheduling Rules

The four acyclic daemon scheduling rules achieve three very important goals: (1) daemons are executed a minimal number of times; (2) daemons can be considered to define invariant relationships between output values; and (3) daemons operate in a stable environment. These goals are further explained below, along with the manner in which they are achieved by the scheduling rules.

The first goal, minimal daemon execution, is guaranteed in two senses:

First, if it is not necessary that a daemon run -- where necessity is defined by changes in the values of its watched outputs -- the daemon will not run at all by virtue of Rule 1, selection. This is achieved on a completely local basis, without reference to part of the picture as "background" and another part as "foreground".

Second, daemon execution is minimal in that if a daemon does run, it is run only <u>once</u> for each set of changes induced by the driving program. This is easily proved: If a daemon D runs twice, then, by Rule 1, one of D's watched outputs must have changed after the first run.

Now, D's watched outputs can only be changed by one of his web fathers. Since, by Rule 2, noninterruption, D's running and his father's running cannot be embedded in each other, D's father must have run after D; but this means that Rule 3, ancestors first, has been violated. Hence D could not have been run twice.

There is an exception to the above: a daemon initially run by ONS or CONTIN may run a second time. This arises due to the fact that picture function body execution violates Rule 2, as discussed in the preceding section; so does ONS.

The second goal, that daemons be considered as defining invariant relationships between their watched and specified outputs, is critical to making DALI programs operate in a comprehensible fashion. It means, for example, that whenever and wherever a RELP module is used, that module's output can be reliably used in place of the sum of its inputs' values.

This second "relationship" goal is primarily achieved by Rule 3, ancestors first. This is so because Rule 3 guarantees that a daemon will be able to run and adjust its specified outputs in accordance with its watched outputs before anyone uses its specified outputs. An instance of this is illustrated in Fig. 3-8a: if daemon A runs and OUCHes its output, Rule 3 guarantees that daemon B runs before daemon C; thus C can rely upon B's maintaining some relationship between its output and A's.

Rule 2, noninterruption, is also needed if daemons are to be invariant relationships. This is the case because a daemon can specify several outputs, holding those outputs in some fixed mutual relationship, and still take an arbitrarily long time between separately adjusting their values. Fig. 3-8b illustrates this.

The third goal, that daemons should operate in a stable environment, is taken to mean the following: the only data value changes affecting a daemon's operation which occur during that daemon's

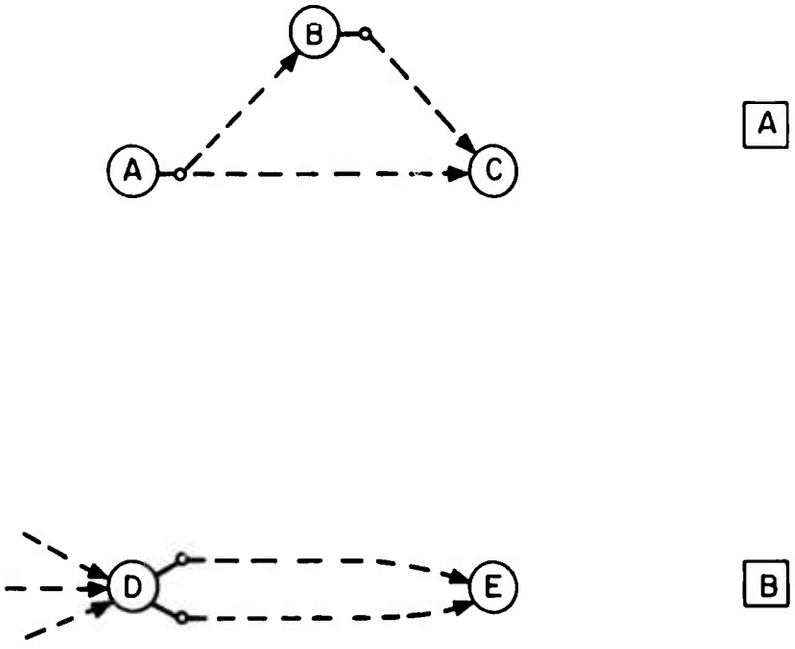


FIGURE 3-8 MODULES AS RELATIONSHIPS

execution are performed by the daemon itself. This is achieved by Rule 2, noninterruption. This could be called the "single instruction sequence" goal: it ensures that despite the fact that DALI is a multiple-environment, "multiple-process" system, each individual daemon body code sequence can be written as if it were running on the simplest possible single program counter machine lacking any form of multiple processing or interrupts. Races, hazards, deadlocks, and other synchronization problems are defined out of existence from the start. It should be noted that despite this, Rules 2 and 3 do provide for true parallel processing of daemons unrelated in web ancestry.

In its use of Rule 2, noninterruption, DALI takes an approach to the problem of inter-process synchronization which differs from that normally taken in, for example, operating system design -- e.g., [Dij1, Hab1]. The normal approach, which is appropriate to the circumstances under which it is being used, is to initially assume truly parallel operation, with the concomitant fact that relative scheduling of operations will be arbitrary and generally malicious. Then an attempt is made to impose some order on the resultant chaos through the use of simple arbiters, semaphores, and the like. In contrast, the somewhat more special-purpose nature of DALI allows it to impose from the very start an extremely strict organization on when and how separate daemons ("processes") may run, allowing parallelism only when it manifestly can do no harm. This dichotomy between DALI and the usual approach has an analog in classical finite-state machine design: DALI corresponds to a synchronous machine rather than to an asynchronous machine.

The last two goals -- daemons are relationships and operate in a stable environment -- contribute heavily to the usability of DALI. It has been the author's sad experience that in systems providing pseudo-parallel processing -- e.g., via non-local GOTOs or inter-process RESUMEs -- the "bugs" that (always!) occur due to inter-process

interactions are severe enough to have a corrosive effect on a programmer's sanity. Furthermore, this occurs even when the total number of processes is very small, like two. A system like DALI, where use of literally hundreds of "processes" -- modules -- is implied, would be totally unusable if such "bugs" could occur with their usual frequency. No small part of the design of DALI was motivated by a desire to create inter-element communication and control-switching mechanisms whose operation would allow truly straightforward, even simpleminded programming.

3.10 External Interrupts

An interesting question which concerns the scheduling rules is how to treat interrupts from external devices, including in particular buttons, tablets, light pens, etc. Although the reported work is not primarily concerned with graphic input, as was stated in section 1.2, this issue is important enough to warrant some discussion.

In the context of DALI, external input devices are best treated as the specifiers of outputs, and interrupts from such devices are best treated as OUCHes of those outputs; e.g., a button has an associated output with a boolean value, OUCHed to *true* when the button is depressed and OUCHed to *false* when the button is released. Such an OUCH will be called an *external OUCH*. It is assumed that a low-level interrupt handler for a physical device will, in a manner totally invisible to user programs, "really" interrupt the daemon currently running when the external interrupt occurs and cause the associated external OUCH to be performed at some time, not necessarily as an immediate part of handling the interrupt. It should be noted that "the daemon currently running" does include the driving process. The primary issue is this: exactly when is the external OUCH performed? I.e., when is the output's value changed and the watching daemons queued?

There are only two absolutely safe times for performing external OUCHes: just as a driving process' call of UPDATE-DISPLAY is entered, and just before it is exited; recall that UPDATE-DISPLAY, discussed at the end of section 3.8, causes control to leave the driving process and enter the picture definition. The reason the entry and exit points of UPDATE-DISPLAY are "safe" is that only then are we guaranteed that both the data specified by the driving process, and all the output values within the picture definition, are in as consistent a state as the user has provided for; it is a useful characteristic of DALI that such "safe" times exist at all.

Performing external OUCHes as part of UPDATE-DISPLAY's entry and exit is equivalent to considering external interrupts as deriving from the driving process' actions, and thereby separating them completely from the construction of the picture itself; this is a point of view which has previously been espoused by Newman and Sproull [New2, New3].

It might seem that performing external OUCHes only as a part of UPDATE-DISPLAY would lead to unacceptable delays between user actions and responses to them; actually, this is not the case. When interaction is actively occurring, the driving process will most usually be in a tight loop, doing little except repetitive UPDATE-DISPLAYs; this loop need not, of course, be active: an "UPDATE-HANG" could be used to put the physical processor into a quiescent state from which it emerges to do UPDATE-DISPLAYs whenever an external OUCH occurs. If the total response of the picture definition to an external OUCH takes too long for reasonable interaction -- e.g., the value changes get propagated into a daemon which does a fourth-order Runge-Kutta integration before returning -- then the programmer has simply overloaded the system.

There is only one other reasonable time to perform external OUCHes: between executions of queued daemons. This can cut short the calculation of the current crop of display changes by queueing daemons of "higher" priority, i.e., of lower numerical daemon priority values as defined in section 3.8. This situation can be envisioned as halting a

wave of computation which is sweeping forward through the data web and re-starting the wave at some point which was previously passed. There are two cases to be considered here: first, the case where the interrupted wave of computation is associated with a previous identical interrupt, e.g., two pushes of the same button in short succession; second, the case where the interrupted wave of computation did not have anything to do with the interrupt, e.g., some large computation is being performed and a pen-tracking cross is to be moved. These two cases are discussed below.

If the initial wave of computation was associated with a previous similar interrupt, performing the external OUCH between daemon executions is simply useless. This is the case because the daemons actually changing the display file will usually be found at the outer fringes of the data web, and so they will never run if this "re-start" interruption occurs frequently enough: despite all the work being done, the picture never changes. Furthermore, as a result of the interruption the outputs internal to the data web may be in such a state that the succeeding wave will find them inconsistent. Performing external OUCHes only on UPDATE-DISPLAY, which amounts to letting each wave of computation finish before starting the next one, may result in slower motion in some cases; but at least the display will change, and the feedback its speed provides may prompt the operator to slow down to a point where the system can track his actions.

If the interrupted wave of computation was not closely associated with the interaction, the situation is somewhat different and may result in an action closer to the classical situation of interrupt handling: the large wave of computation halts, another smaller wave is begun, finishes, and the large wave continues from where it left off. Here we effectively have two separate pictures being produced which "just happen" to appear on the same display screen. There is no reason why this should not be done; but as in the first case, any interaction between the two waves may result in mutually inconsistent output values

when the large wave starts up again. The possibility of such inconsistencies can be avoided if the "two pictures" are considered as two entirely separate DALI systems, and communicate only through mutually external OUCHes that occur for each of them at the times they enter and exit their respective UPDATE-DISPLAYs.

3.11 The Total Environment

The local environment is not the only identifier-to-value mapping available to a running daemon; for example, the value of the identifier "+" must be obtained from somewhere. The complete set of such mappings accessible to a daemon is called the total environment, and is described in this section.

The term "body" shall be used below to indicate either a daemon body or a picture function body.

The total environment for the evaluation of a body is composed of four parts:

(1) the temporary environment

(2) the local environment

(3) the global environment

(4) the primitive environment.

The characteristics of these four are summarized in Fig. 3-9, and will now be discussed.

The temporary environment is a purely temporary storage environment. It contains, e.g., PROG variables and intermediate expression results, and is created as needed by a body. It is empty when body evaluation starts, and is emptied when it stops. Each temporary environment is unique to a given body evaluation; it is not retained across separate runs of a given daemon.

The local environment is that element of a picture module of the

CHARACTERISTIC / ENVIRONMENT	SIZE	LIFESPAN	ACCESS	WRITEABLE?
TEMPORARY	VARIABLE	BODY EVALUATION	A BODY	YES
LOCAL	FIXED	LIFE OF MODULE	A MODULE	YES
GLOBAL	FIXED	LIFE OF PROGRAM	EVERYBODY	YES
PRIMITIVE	FIXED	LIFE OF IMPLEMENTATION	EVERYBODY	NO*

* FOR INTERPRETED LISP, "YES."

FIGURE 3-9 SUMMARY OF ENVIRONMENT CHARACTERISTICS

same name, and has been previously described. Its salient characteristics are that (1) it is unique to a given picture module, and hence to a given group of daemons; and (2) its contents are retained across body evaluations. It thus provides groups of daemons with private memory.

The global environment is accessible by bodies according to the static block structure of the base language. In LISP, it is an undifferentiated whole, totally accessible by every body. Its purpose is to provide bodies with a shared pool of memory whose contents may vary across programs; in particular, it gives bodies access to user-defined picture function definitions. It is unique to a given program, and its structure is fixed through all execution. Its contents may be changed by assignment.

The primitive environment contains the definitions of DALI and base language primitives. In most base language systems, but not interpreted LISP, it is by nature read-only. The LISP exception aside, it is unique to a given language implementation.

These four environments may be considered as arranged in a stack, as shown in Fig. 3-10. Identifier lookup conceptually follows the stack from temporary to primitive environments.

The total environment structure formed by many picture modules in a totally quiescent state is thus shallow but broad, having a "local environment branch" for each picture module. This is depicted in Fig. 3-11a.

The situation when DALI execution is in progress is separable, for clarity, into two cases: (1) structurally non-additive change, i.e., picture functions are not being applied; and (2) structurally additive change, i.e., picture function application is taking place.

In both cases, the unique daemon which is the driving program is active, and hence has a temporary environment.

In the structurally non-additive case, shown in Fig. 3-11b, only

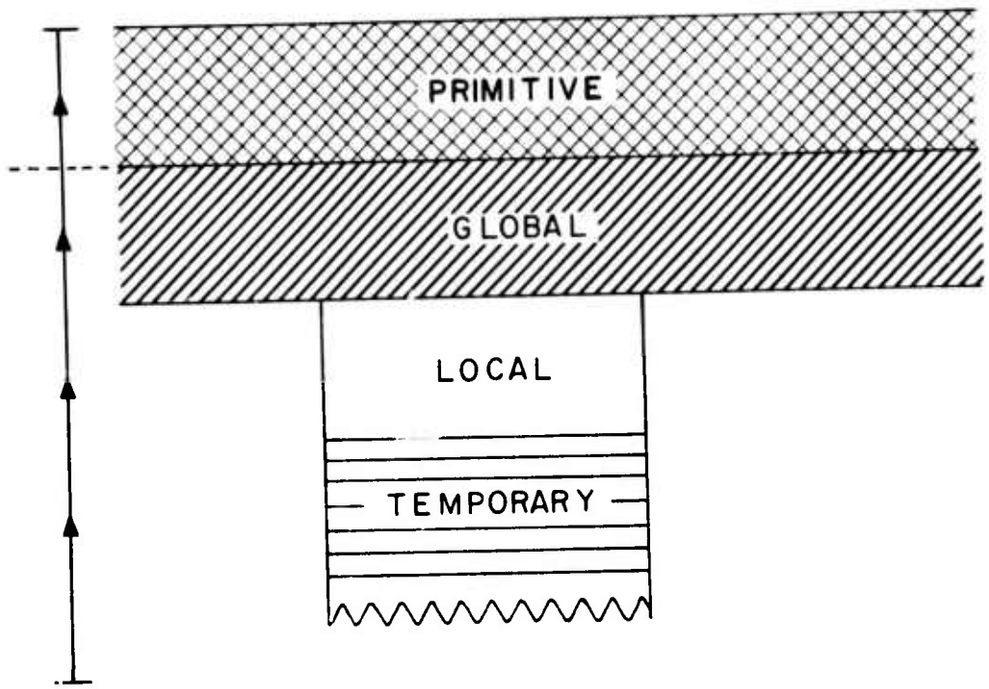


FIGURE 3-10 ENVIRONMENT FOR BODY EVALUATION

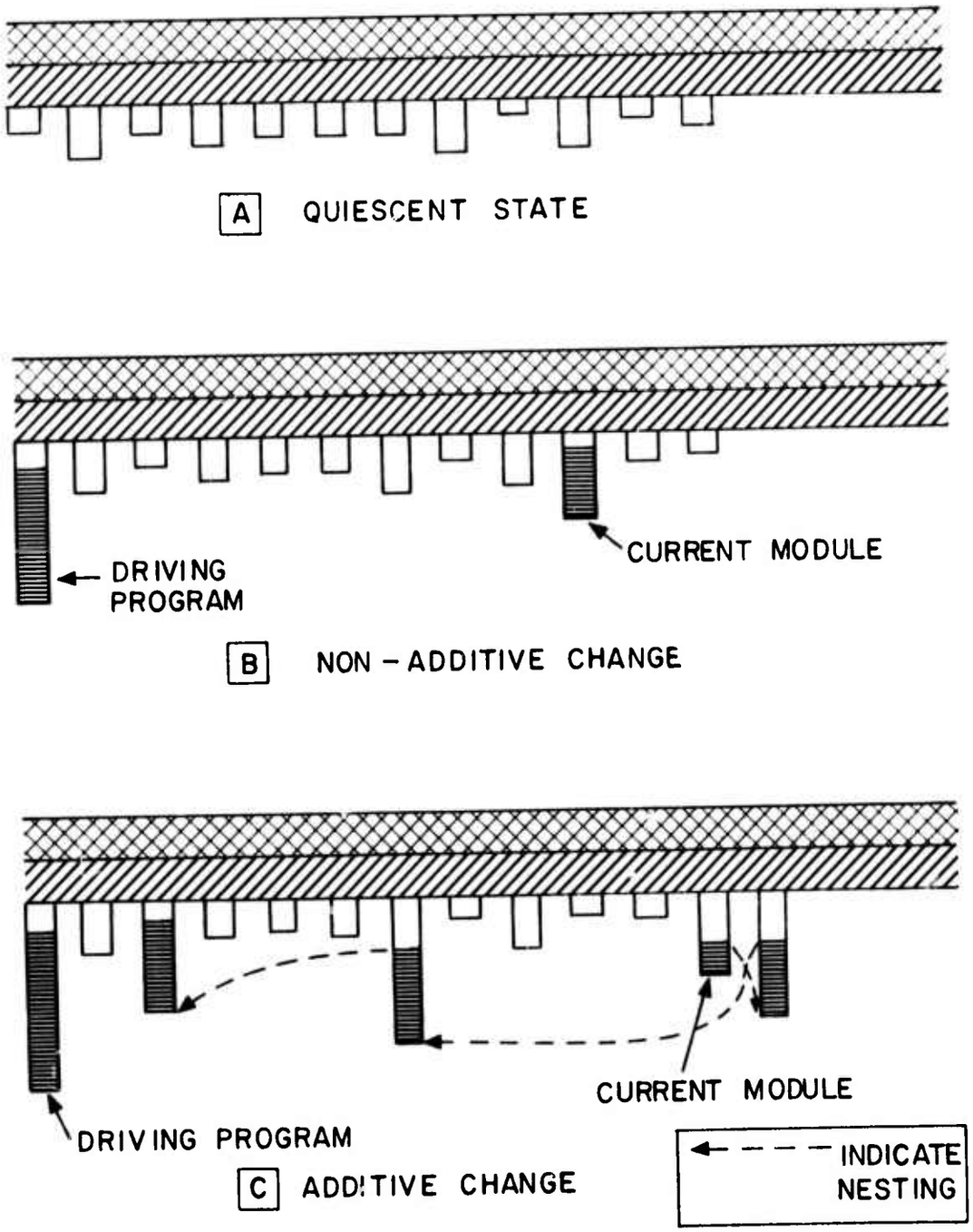


FIGURE 3 – II LOGICAL ENVIRONMENT STRUCTURE

one other body will be active, namely that of the daemon currently running; and hence only one other temporary environment will exist. This is a consequence of the fact that a daemon's body evaluation is not interrupted.

Several daemon bodies may be active in the structurally additive case, but their activity will be strictly nested. This nesting was previously mentioned in section 3.7, and is illustrated in Fig. 3-11c.

While the above describes the logical situation, the actual situation is simpler from the point of view of storage management: Due to the fact that the only multiple activity is nested, all temporary environments may be held in a single stack. This is shown in Fig. 3-12.

The organization of the total environment as described is stack-oriented, with local environment blocks, each fixed in size, held in a heap. Identifier evaluation is quite simple and efficient with this organization: (1) primitive environment identifiers are effectively evaluated at compile time; (2) global identifiers' values are held in static memory locations; (3) local environment identifiers' values are found by indexing from a "current local environment" state variable; and (4) temporary environment identifiers values are found by indexing from a "current stack top" state variable. The context switching required when entering a body thus consists simply of swapping in a new "current local environment" state variable. Interpretive implementations, as opposed to compiled ones, could operate in a similar fashion.

The temporary environment could be static as would be the case if FORTRAN were the basc language. This would rule out recursive picture functions; but it would not rule out multiple modules from the same picture function, since the local environment holds the information local to each module and only one non-picture-function-body daemon is running at any given time.

An alternative organization could allow each daemon to retain a temporary environment; in this case, running a daemon could "resume" its

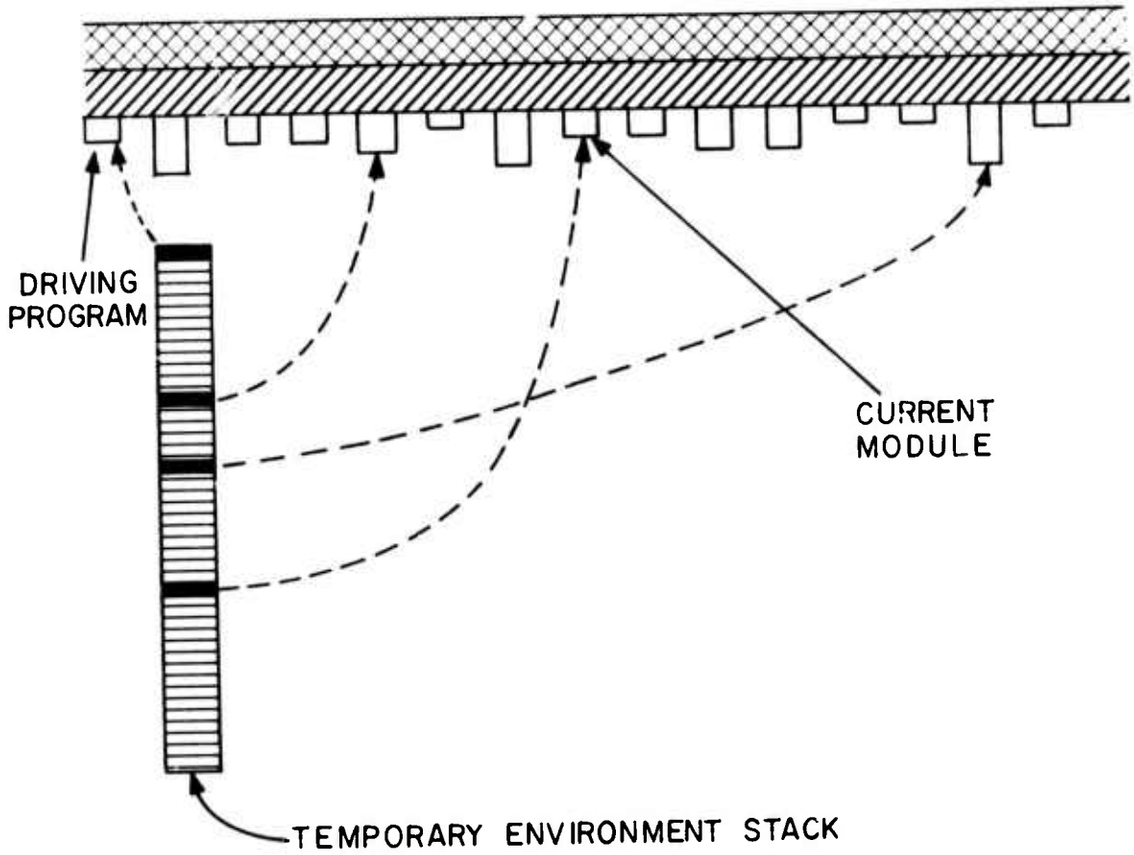


FIGURE 3-12 STACKED STORAGE OF TEMPORARY ENVIRONMENTS

execution wherever it left off. While the resulting system might be more flexible for certain cases, much more active use of heap storage would be required to record each daemon's temporary environment while that daemon was quiescent.

DALI utilizes a bipartite environment structure, with the temporary, global, and primitive environments held -- for purposes of this discussion -- on a stack, and the local environment held in "heap" storage. This is a compromise between fully stack-structured conrol and environment schemes, epitomized by ALGOL 60, and fully tree-structures schemes, as for example used by CONNIVER [McD1], OREGANO [Ber1], and SIMULA 67 [Dah2]. A comparison of DALI's scheme with the other two is interesting and will be pursued here.

Relative to stack-structured schemes, or, for that matter, static schemes such as FORTRAN's, DALI's bipartite scheme is less efficient because it requires active use of heap storage. However, use of a heap appears to be a requirement of any system aimed at constructing changing pictures; so this cannot be considered a major disadvantage.

DALI's distinction between environment layers that can be reclaimed when they are exited -- the temporary environment -- and environment layers that always stay in existence until explicitly destroyed (or garbage colleted) -- the local environment -- provides more efficiency than fully tree-structured schemes, both in execution time and in storage space; however, it sacrifices some flexibility.

The advantage in execution time arises primarily from two sources. First, as noted earlier in this section, identifier evaluation can be simple in DALI; it never intrinsically involves searching. Second, storage allocation is simple. The latter is true because retained local environments, held in the heap, are created and destroyed relatively seldom; while the actively varying temporary environment is easily allocated on a stack.

Some advantage in storage space accrues from the fact that

temporary environment layers are known a priori to be stack-allocated, and so can generally be simpler than layers destined for retention. This also provides some added execution time savings, since less complex layers take less time to build.

However, the primary spatial advantage of DALI over fully tree-structured schemes arises from the fact that no matter how deeply nested in temporary environment layers a retained layer is, i.e., how deeply a picture function application is nested, the entire temporary environment is fully reclaimed and its storage immediately re-used. In a fully tree-structured scheme, all the "temporary" layers in which a retained environment is nested, e.g., block entries, layers of conditional evaluation, etc., are retained along with the "desired" layer or layers. If the ability to exit multiple times out of any already-exited procedure is desired, this global retention is needed. But this is not truly necessary if -- as in DALI daemons, CONNIVER "methods", and the majority of the examples the author has seen of SIMULA's use of this feature -- the desired action is to enter a "saved" environment, do some local processing, and exit to an environment not that of the original caller. Of course, if the only environmental data available is in the tree-structure, the local processing necessarily makes use of the layers "above" the currently active one. This is not the case in DALI; daemons are "sealed off" from their containment tree ancestors' retained environments fairly thoroughly, a situation made possible because daemons have the global and primitive environments available.

Nevertheless, flexibility is sacrificed in DALI's scheme, shown by the fact that in the scheme as presented, it is not possible for a DALI daemon to suspend itself midway through its code and then return to the suspension point at the next invocation of that daemon. Use of such an ability can, in some cases, result in a simplification of user code; an example in which this is true is the interactive drawing program to be presented in section 3.12, which could have been written in a somewhat simpler fashion using this technique. As was pointed out earlier in

this section, this capability could be produced by allowing some form of special exit from a daemon which causes the "temporary" environment at the time of exit to be retained until the next invocation of the daemon. This was not done for three reasons: First, its lack makes DALI a somewhat less drastic language extension, and hence easier to implement. Second, use of this feature implies much more active and time-consuming use of heap storage, "spagetti stacks", or other complex storage allocation schemes. Third, the author could not find any examples where such a facility provided a truly substantial advantage. All of these reasons are perhaps specious: it may have been the better choice to allow retention of daemon's "temporary" environments. This may particularly be the case with respect to interactive input, since, on the evidence of Newman's finite-state "Reaction Handler" system [New2], the problem of parsing operator actions as a "command language" may possibly be fruitfully approached as a use of co-routines with the operator considered as one of the co-routines.

But the retention of "temporary" environments is still not a completely tree-structured environment system, since distinction is still maintained between picture functions and "normal" functions: the former build a (containment) tree structure of local environments, and the latter can, via some new form of "daemon exit", retain only a single control and environment point on each daemon's "temporary" stack. The author does not feel that a fully tree-structured control and environment scheme is either necessary or desirable, for the reasons noted below.

To some extent, the issue is one of efficiency since, as noted above, there seems to be little real use for many of the retained enviroment layers in which a particular desired group of environment layers is nested. It is also an issue of modularity: the sealing off of a module from its fathers promotes more conscious consideration of the interfaces between objects.

The primary issue, however, is this: Examples of code in which a

procedure literally "returns twice" to its original caller, and hence makes direct use of all the retained environment and control layers, very often appear to be extraordinarily compact expressions of the desired action; for example, the author has seen a 12-line "time-sharing system", i.e., a clock-driven process-swapping system, written in CONNIVER. But at the same time, these examples uniformly appear to the author to be extraordinarily complex, confusing, difficult to follow, and hence "bug-prone". Since one of the goals of this work is the creation of a system which is as straightforward to use as possible, constructs whose effects are difficult to comprehend are to be distinctly avoided.

Such difficulties of comprehension may simply reflect a deficiency of the author; but they may be an indication that, not unlike the GOTO, constructs such as the ability to "return twice" should be declared anathema even at the high level of sophistication at which they are normally used. In fact, the level of sophistication required for the use of such constructs is in itself an argument in this direction, since it implies that an unwarranted amount of intellectual effort needs to be spent on details of implementation. This should not be taken as an advocation of the use of DALI's particular scheme in more general circumstances; rather, it is an observation that the full generality of tree-structured control and environment structures apparently contains a source of confusion which does not really seem to be an intrinsic part of the capabilities truly desired.

3.12 More Examples: Iteration, DODA, and Input

To complete the example of a bar graph which was originally mentioned in section 2.3, a picture function BARGRAPH, which draws a full bar graph reflecting the state of a set of values, is given below.

The picture function ONEBAR, used by BARGRAPH to draw individual bars of the graph, is also given.

```
(DEFPIC BARGRAPH (VALS MAX LB RT
                  "AUXO" RB SCL WIDTH "AUX" NVALS)
   (SETQ NVALS (LENGTH VALS))
   (CONTIN (OUCH RB (POS (X ,RT) (Y ,LB)))
           (OUCH WIDTH (POS (/ (- (X ,RT) (X ,LB)) NVALS) 0))
           (OUCH SCL (/ (- (Y ,RT) (Y ,LB)) ,MAX)) )
   (PROG (PREV)
      (SETQ PREV LB)
      (MAPC (LAMBDA (V)
              (SETQ PREV !(ONEBAR V PREV SCL WIDTH)))
            VALS)
      (LINE PREV RB)))

(DEFPIC ONEBAR (V PREV SCL WD "OUTU" RP "AUXO" LP)
   (CONTIN (OUCH LP (POS (X ,PREV) (* ,V ,SCL))))
   (SETQ RP !(RELINE LP WD))
   (LINE PREV LP))
```

Fig. 3-13 illustrates the desired graph for 5 values, and Fig. 3-14 illustrates the "subpicture" drawn by each ONEBAR.

The data web produced by BARGRAPH when graphing two values is shown in Fig. 3-15a, labelled as illustrated in Fig. 3-15b.

BARGRAPH takes four arguments: (1) VALS, a list of outputs which hold the represented values; VALS is not itself an output, and the number of entries in the bar graph is not variable. (2) MAX, the maximum expected value, used to establish the appropriate scaling; it is an output, but its value will probably not change as often as the individual outputs. MAX is not necessarily a running maximum of the values presented, although it could be. (3) and (4), LB and RT, position-valued outputs specifying the Left Bottom and Right Top of the area in which the graph is to appear.

BARGRAPH also maintains three "AUXO" outputs: (1) RB, the right-bottom corner of the area of the graph used to draw the final rightmost edge; (2) SCL, the scale factor used to map values into displayed heights; and (3) WIDTH, the width of each BAR. An "AUX" variable, NVALS, holds the number of values so that re-computing the length of VALS is unnecessary.

The (MAPC . . .) construction in BARGRAPH's PROG applies the unnamed function (LAMBDA (V) . . .) to each element of VALS in

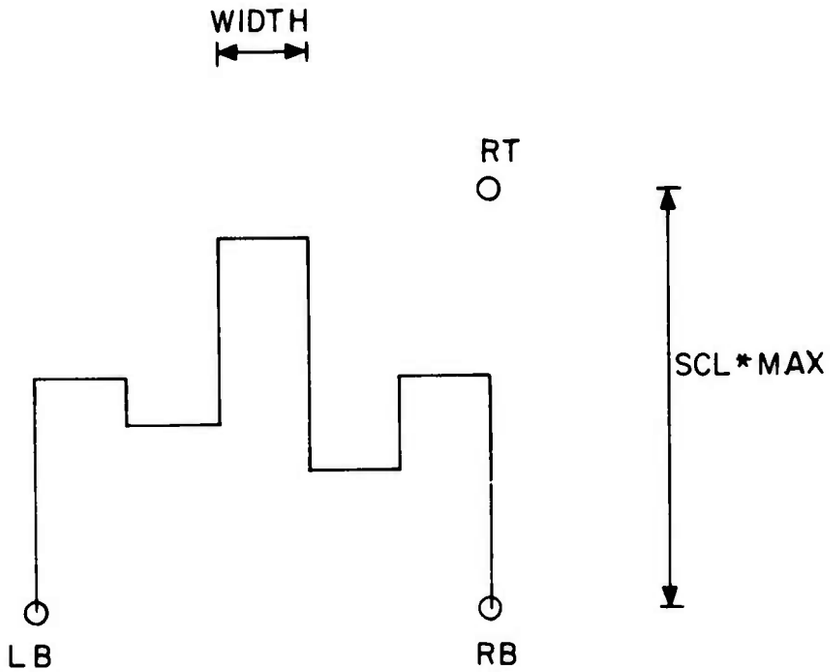


FIGURE 3-13 DESIRED BAR GRAPH

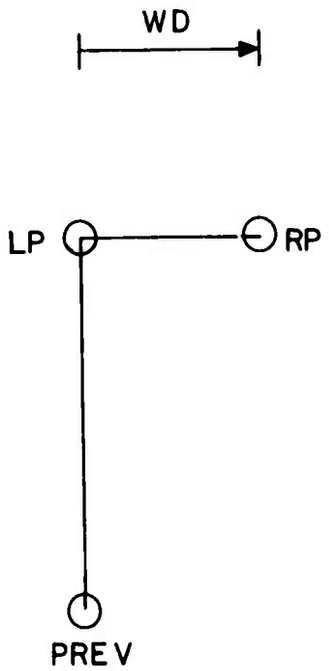
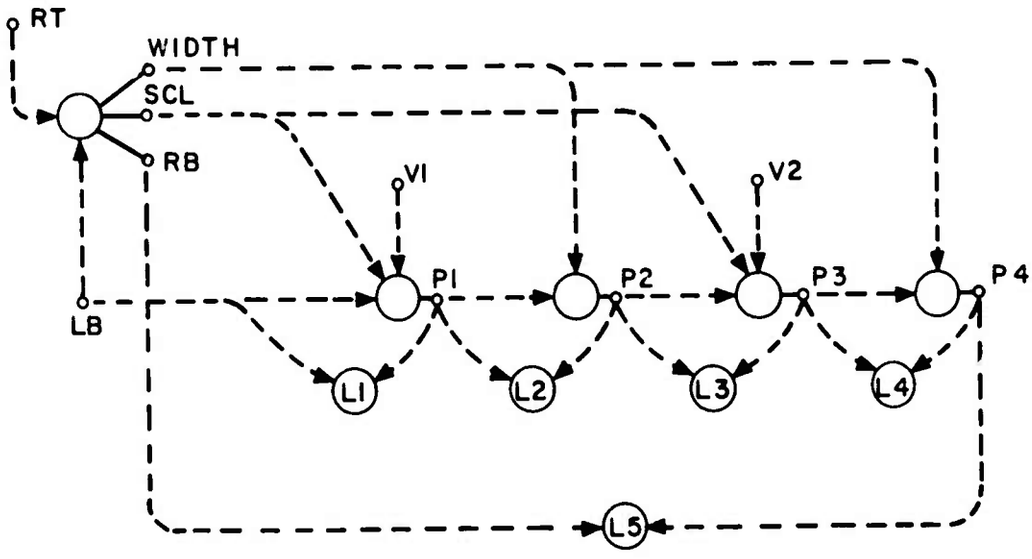
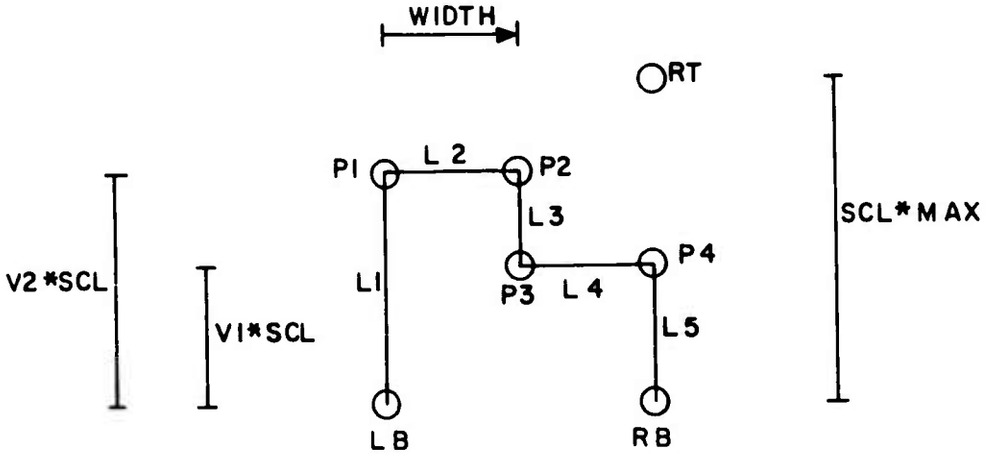


FIGURE 3-14 ONEBAR VARIABLES

A





B

FIGURE 3-15 BARGRAPH

succession. Successive ONEBARs are given a PREV argument which is the output of the preceding ONEBAR; this allows appropriate vertical bar sides to be drawn. Note that the variable PREV of BARGRAPH's PROG and the variable V in the unnamed function are in BARGRAPH's temporary environment and hence have values only during the execution of the PROG when BARGRAPH is initially called.

ONEBAR is very similar to the BAR example of Fig. 3.7, but is simpler. It only draws the left and top sides of a bar; and it need not construct a suitable width vector, as this task is performed by BARGRAPH for all the ONEBARs.

It would appear from Fig. 3-15 that a change in one of the values would cause every daemon controlling lines to the right of that value to be run; for example, a change in Fig. 3-15's V1 would cause the running of L1 through L5 along with the daemons specifying P1 through P4. This is not the case. Changing V1, for example, causes P1 and P2 to change; this is necessary. The P2 change will cause the daemon specifying P3 to also be run. However, the OUCH of P3 will not cause P3's value to change, since P3 depends only on P2's X component and a V1 change will affect only P2's Y component. Thus, by the definition of OUCH in section 3.3, the P3 OUCH has no effect and P4, L4, and L5 will not run.

As an example of how DALI might be fit into a more conventional infix-oriented language, BARGRAPH and ONEBAR are coded below in a hypothetical DALI extension of EULER [Wir1] (called DEULI?); EULER is itself an extension of ALGOL.

```
picfun BARGRAPH input VALS, MAX, LB, RT
               auxo RB, SCL, WIDTH aux NVALS;
begin NVALS:=length[VALS];
      contin RB:::=[x[.RB],y[.LB]};
             WIDTH:::=(x[.RT]-x[.LB])/NVALS,0};
             SCL:::=(y[.RT]-y[.LB])/NVALS
      end;
      begin new PREV, IX;
            PREV:= LB;
            for IX:=1,1,NVALS do
                PREV:=!ONEBAR[VALS[IX],PREV,SCL,WIDTH];
            line[PREV,RB]
      end;
end;
```

```
picfun ONEBAR input V, PREV, SCL, WD outu RP auxo LP;
begin contin LP:::{x[.PREV],.V*.SCL} end;
      RP:=!reline[LP,WD];
      line[PREV,LP]
end;
```

Conventions used in the above DEULI code are: reserved words and system functions, including line and reline, are in lower case and underlined; period (.), rather than comma, indicates an application of OVAL; braces ({}) make their contents into a position; the infix operator ::: does an OUCH of its left operand to the value computed by its right operand; contin implies a begin and must be closed with an end; and as with DALI in LISP, a prefixed exclamation point indicates an application of OUT.

Before proceeding on to more examples, the DALI primitive for dynamic structural iteration will be described. It is a picture function called DODA, for DO in DAli, a name pronounced as in the song "Camptown Racers". The form of DODA's iteration is suggested by the previous BARGRAPH example: it creates a variable-length data web chain of picture modules, as illustrated in Fig. 3-16.

An application of DODA has this form:

(DODA n (-initials-) obj) .

n is an output whose value, a non-negative integer, is the number of modules DODA has chained together at any given time. When n increases, modules are created and placed on the end of the chain one by one; when n decreases, modules are deleted from the end of the chain. obj is some object which returns a picture module when it is evaluated as if it were a one-statement daemon owned by the owner of the DODA's caller; such evaluation is how DODA creates modules. (-initials-) is used in the chaining operation.

The chain is formed by use of the function PREOUT. PREOUT can be applied legally only in the obj argument of a DODA, or any normal function or procedure -- not picture function -- called by that argument. PREOUT takes one argument an integer greater than 0. An application of PREOUT,

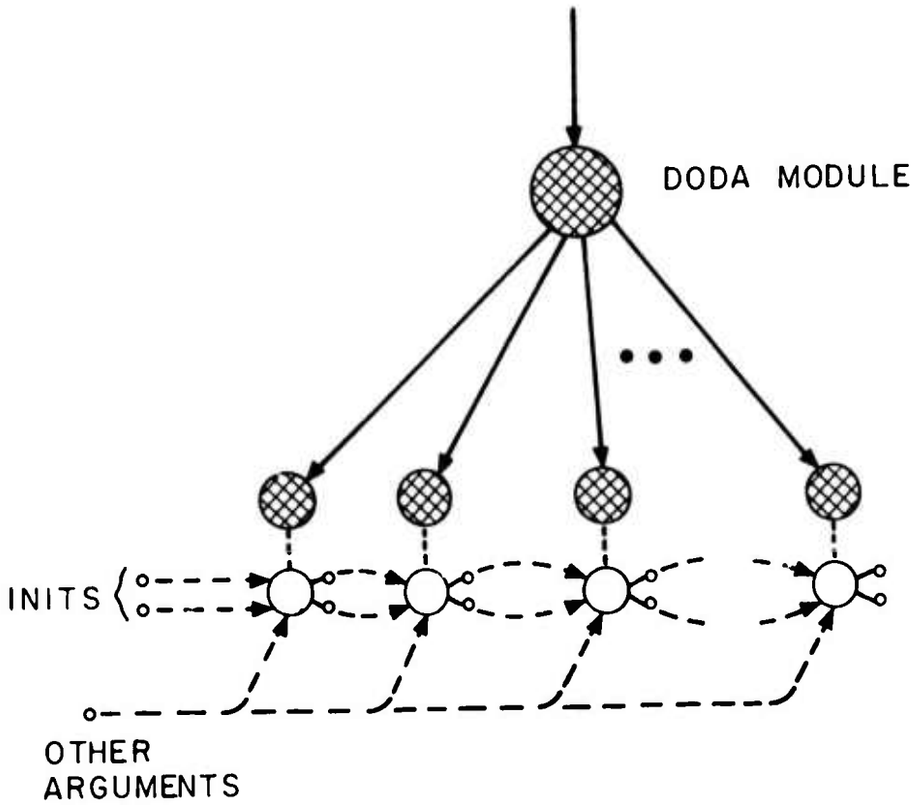


FIGURE 3-16 DODA PICTURE STRUCTURE

(PREOUT m)

returns the mth output of the previous menber of the chain. If there is no previous member -- i.e., obj is being used to construct the chain's first member -- the mth element of DODA's (-initials-) argument is returned instead.

DODA could be defined as a user-written function in a purely interpretive version of DALI, but it would require primitives not presented here. In particular, some method of making the number of module outputs variable at module creation time would be needed. Provision would also have to be made for unevaluated arguments, and a DALI EVAL function would be necessary. Furthermore, PREOUT must violate the normal environment discipline to find the currently operating DODA's list of chained outputs. The environment structure existing when DODA is creating a module, complete with the "illegal" reference required by PREOUT, is shown in Fig. 3-17.

A rectangular grid drawing picture function, GRID, will now be defined using DODA:

```
(DEFPIC GRID (NX NY HT WD LL "AUXO" DELX DELY)
    (CONTIN (OUCH DELY (POS 0 (/ (Y ,HT) (- ,NY 1)))))
    (CONTIN (OUCH DELX (POS (/ (X ,WD) (- ,NX 1)) 0)))
    (DODA NX (LL) (GRIDDLE (PREOUT 1) HT DELX))
    (DODA NY (LL) (GRIDDLE (PREOUT 1) WD DELY)) )

(DEFPIC GRIDDLE (BOTPOS LNTH DEL "OUT" NEXT)
    (CONTIN (OUCH NEXT (+ ,BOTPOS ,DEL)) )
    (RELINE BOTPOS LNTH) )
```

GRID's arguments are all outputs, and allow variation in the grid's height and width (HT and WD), the number of lines drawn in the X and Y directions (NX and NY), and the position of the lower left corner (LL). NX and NY have integer values, and the values of HT and WD are assumed to be of the form (POS 0 height) and (POS width 0) respectively. LL's value is also a position. DELX and DELY are the horizontal and vertical vector distances between successive vertical and horizontal lines, respectively.

GRIDDLE is used to actually draw the lines. Each GRIDDLE module

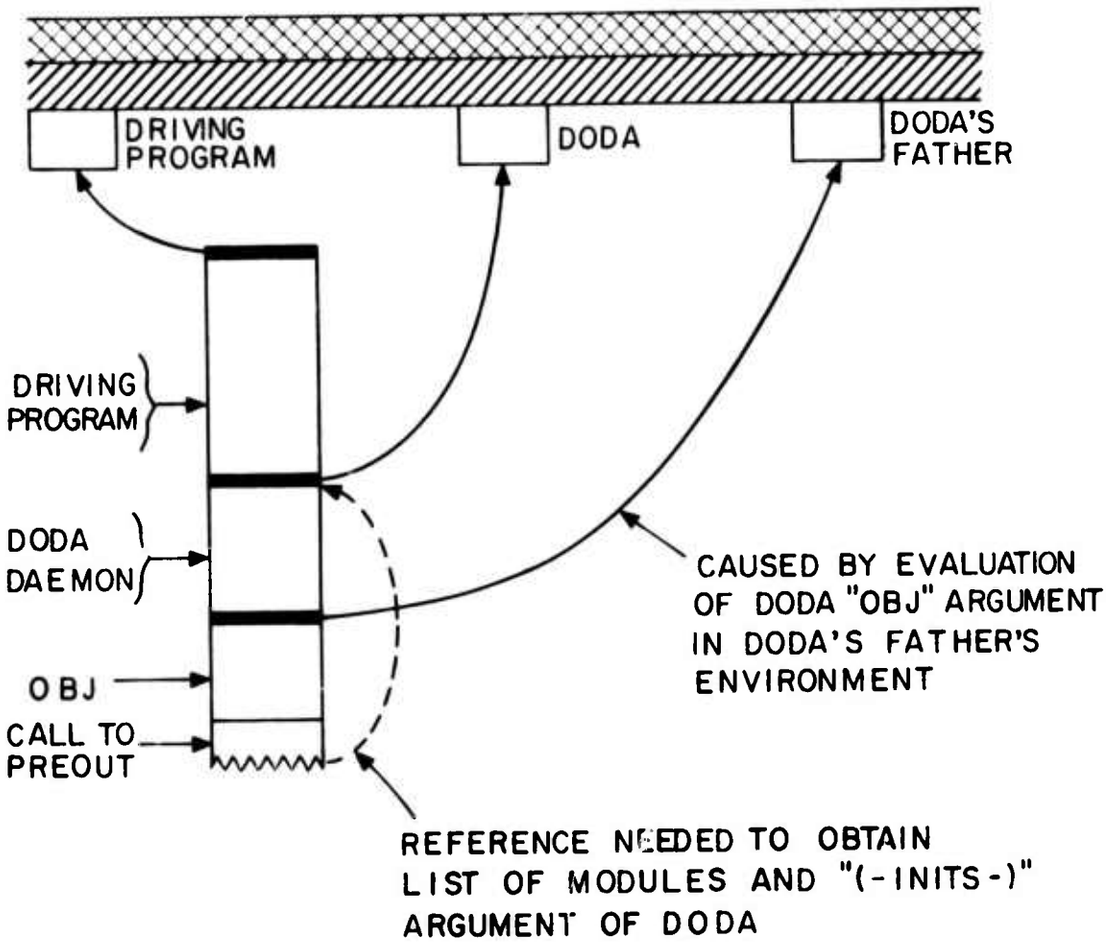


FIGURE 3-17 DODA CREATING A MODULE

feeds his successor in the chain the position at which the successor should start his line. Note that changes to LL result in response only by the daemons in GRIDDLE and its sons.

The next example is a regular polygon whose number of sides, radius, and center position can be varied. This is NGON, below, in which N is the number of sides, C is the center, and R is the radius. DTHET is an output whose value is held at the change in angle between two corners.

```
(DEFPIC NGON (N C R "AUXO" DTHET SP)
    (CONTIN (OUCH DTHET (/ 6.2832 ,N)))
    (CONTIN (OUCH SP (+ ,C (POS ,R 0))) )
    (DODA N (SP 0) (CORNER (PREOUT 1) (PREOUT 2))) )

(DEFPIC CORNER (PP THET "EXTERNAL" R C DTHET
                       "OUT" NP NTHET)
    (CONTIN (OUCH NTHET (+ ,THET ,DTHET)))
    (CONTIN (OUCH NP (+ ,C (* ,R (POS (SIN ,NTHET)
                                      (COS ,NTHET))))))
    (LINE PP NP) )
```

Each CORNER uses the previous corner's position (PP) and the angle a radial vector to that corner makes with the X-axis (THET). It computes the next angle (NTHET) for its successor, along with its successor's corner position (NP). Finally, it causes a line to be drawn between adjacent corners. NGON itself creates and updates the position of the initial corner (SP).

The final example is a simple interactive drawing program, DRAW, whose code is below. We assume here the existence of three external outputs of the type discussed in section 3.10; they are given to DRAW as arguments. The value of the first, PPOS, is the position of some input device like a tablet pen. The second, PBUT, has a boolean value associated with a switch on the pen; it is <u>true</u> (T) when the switch is depressed, and <u>false</u> (NIL) otherwise. The third, RESET, is likewise a button boolean. DRAW assumes a very simple driving program loop which continuously does UPDATE-DISPLAYs, as discussed in section 3.10.

When the PBUT switch is depressed, DRAW alternately (1) remembers

the pen position and (2) creates a line drawn from the remembered position to the current pen position. When the RESET button is depressed, all drawn lines are deleted and the DRAW is reset. We will also draw a cursor dot (zero-length line) at the position of the pen.

The final lines drawn are static, so they will be created with a picture function called STATLINE rather than with LINE. STATLINE takes constant position arguments rather than outputs and lacks a daemon to change the display file entry; otherwise it is identical to LINE. In particular, STATLINE does have LINE's DELETE daemon.

```
(DEFPIC DRAW (PPOS PBUT RESET "AUX" LASTPT STATE LINES)
   (SETQ STATE T)                          ;Initialize state flag
   (SETQ LINES ())                            ;and list of lines.
   (LINE PPOS PPOS)                             ;Draw the cursor.
 ;When the pen button is depressed (PBUT changes to true):
 ;If in record state, record position and switch STATE to NIL.
 ;Otherwise, add to LINES list and switch STATE to T.
   (ONC (VAL PBUT) ()
     (COND (,PBUT
         (COND (STATE (SETQ LASTPT ,PPOS)
                      (SETQ STATE NIL))
               (T (SETQ LINES
                        (CONS (STATLINE LASTPT ,PPOS)
                              LINES))
                  (SETQ STATE T))))))
 ;Whenever the reset button is depressed,
 ;delete all lines and reset STATE to T.
   (AS-NEEDED (COND (,RESET (MAPC DELETE LINES)
                            (SETQ LINES ())
                            (SETQ STATE T)))))
```

Although the record/draw daemon uses PPOS' value, it should run only when the pen button is depressed; hence AS-NEEDED cannot be used.

A DRAW providing a rubber-band line could be written, but it requires primitives not introduced until section 4.4.

3.13 Concluding Notes on Basic M-DALI

Before passing on to more complex issues, two notes are in order.

First, the only primitives for producing visible output which are ever used in this document are LINE, STATLINE, and a picture function

TEXT used in section 4.5 to display a static text string. This has been done for purposes of simplicity; it is unreasonable for a working DALI system because of its excessive storage requirements. Some facility should exist for constructing compact static organizations of lines and dots, analogous to instance tree items, and using them to create visible "instances" as if they were parameterless picture modules. The coordinate system transformation scheme presented in section 4.2 can be used to vary the position, rotation, scale, etc. of such instances. A subsystem operating in a manner analogous to EULER-G [New1] could be used for this purpose.

Second, it is indeed the case that the driving program must create outputs, perform at least one picture function application, and perform OUCHes. This is really too bad; in the best of all possible worlds, a user should be able to tell some omnipotent graphics monitor "I want to see <u>this</u> looking like <u>that</u>" and see it with no modification to the application program. The situation is at least tolerable, however, since a single OUCH can inform the picture definition that the display is to be updated based on a great bundle of information not contained in outputs.

Chapter 4
M-DALI: Further Issues

4.1 Introduction

Unlike most of the other chapters in this document, this chapter lacks a single focus of attention. Instead it deals with five separate topics, each of which, while important, is somewhat peripheral to the main conceptual thrust of this work. These topics are:

(1) How the coordinate transformation schemes characteristic of instance tree systems can be embedded in DALI.

(2) How deletion is accomplished, and why retentive storage management -- "garbage collection" -- is not used instead.

(3) What structural change to the data web requires, and what daemon conditions provide motivation for such structural change.

(4) A discussion of a realistically large example.

(5) In what way "hit" detection can be accomplished in DALI; i.e., how a DALI program can recognize and utilize the fact that an input device such as a tablet is pointing at a particular displayed object.

These five topics are covered in the order listed above. The level of detail in the discussion varies from topic to topic; for example, only the general form of the mechanisms for "hit" detection is indicated, while coordinate transformations are discussed in detail.

4.2 Coordinate Transformations

Coordinate system transformations -- i.e., mappings from the coordinate system in which a visible object is defined to the coordinate system in which it is visibly displayed -- are a basic tool of computer graphics. This section describes how such transformations can be embedded in DALI, first considering the general problem of embedding any transformations at all, and then describing the specific set of two-dimensional transformations -- translation, rotation, scaling, and clipping -- which will be assumed in the remainder of this document. What were referred to as "attributes" in section 1.2, e.g., color, intensity, etc., can be handled in a similar fashion.

As will be explained in this section, a primary characteristic and advantage of DALI in doing coordinate transformations is hardware independence: DALI is capable of utilizing whatever transformation capabilities the available display hardware offers, and supplying whatever capabilities the hardware does not have; the particular mix of hardware and software used can be totally invisible to the user -- except, of course, for the speed with which the system operates.

Associated with each picture module is a <u>total transformation</u>, abbreviated <u>TT</u>. This is a complete set of transformation parameters held as the values of a set of outputs. The TT of a module completely describes the mapping from coordinate data used by daemons of the module to coordinates on the face of the display device used, including, via clipping parameters, how much of the displayed objects are visible.

Total transformations are "the real thing": they alone define how a given set of objects will appear on the display device. The main body of this section is concerned with how total transformations are (1) associated with modules; (2) defined and modified relative to other total transformations; (3) created; (4) accessed by the user; and (5)

implemented. Of particular note among these is the fact that the total transformation parameters can be accessed by user-written code; thus, for example, a daemon can be created which adds or subtracts visible detail from the image displayed in accordance with size changes.

Discussion of each of the five issues listed above follows; as mentioned earlier, the particular transformation system chosen for use in the remainder of this document is covered after these five have been discussed.

A total transformation (TT) is associated with a module when the module is created. This association is permanent; i.e., while the parameter values of a given TT -- the values of the outputs which define the TT -- can be changed, the association between a module and its set of TT outputs cannot be changed.

By default, a picture module shares its TT with its containment tree father. Thus, for example, a LINE picture function called by a daemon creates a module whose associated TT is that of the daemon's owner picture module.

A TT other than the default is associated with a picture module by creating the module using the function TRANSFORM, as in

(TRANSFORM "USING" mod picfun -args-)

This applies the picture function picfun to -args-, and returns the resultant picture module. Associated with the new module is the TT associated with the module mod.

TRANSFORM is also used to create new TTs. This will be discussed later, along with the mechanism behind the association of a module with a TT.

Only one TT initially exists; this is the TT associated with the one initially existing picture module, the root module. This <u>root TT</u> has two unique properties: (1) it cannot be changed; and (2) it is absolute, i.e., not relative to some other TT. All other TTs are

relative, i.e., they are the concatenation of (1) a base TT, which may be the root TT; and (2) an incremental transformation, or IT, specified by the user.

An incremental transformation (IT) is a set of outputs containing transformation parameters, generally defined by some arbitrary user computation. An IT differs from a TT in two ways: (1) It need not be complete; e.g., it might consist only of a single position-valued output used as a translation parameter. (2) An IT can be directly modified by the user; i.e., user-written daemons can, and usually will, specify the outputs which constitute an IT and OUCH them. In contrast to the direct user control of an IT, only by varying the IT(s) concatenated to form a TT can a TT be modified.

The TT resulting from concatenating a given base TT and an IT will be called a resultant TT.

The concatenation operation required to define the parameter values of a TT is conceptually performed by daemons created along with each new TT. These daemons specify a resultant TT's outputs, and they watch the outputs of the base TT and the IT concatenated to produce the desired resultant TT. If the available hardware support is minimal, this may literally be the case; if, however, the available hardware is capable of doing at least part of the job, it can be used. How this is done will be described as part of the description of implementation.

The function TRANSFORM, given a base TT and an IT, creates a new resultant TT. Since, as mentioned, a TT is a set of outputs specified by a set of concatenation daemons, TRANSFORM creates a new picture module to own those outputs and daemons. Such a module is called a transform module, and is said to contain the resultant TT whose outputs it owns. A transform module becomes the containment tree son of the owner of the daemon applying TRANSFORM.

The mechanism behind the association between a module and a TT can now be given: (1) a transform module is associated with the TT it

contains; (2) any non-transform module NT is associated with the TT contained in the transform module closest to NT on the containment tree path from NT to the root module. The root module is a transform module, containing the root TT.

Thus, user access to TT outputs can be achieved via the "EXTERNAL" identifier lookup mechanism described in section 3.11.

The method by which new picture modules are associated with new TTs is straightforward: TRANSFORM is given the picture function and arguments to be applied to create the new module, creates the new module, and makes it a son of an appropriate transform module.

If given both (a) the picture function and arguments, and (b) a base TT and an IT, TRANSFORM both creates a new transform module as the son of its caller and applies the given picture function to create the desired module as a son of the new transform module. In this latter case, TRANSFORM returns the module whose picture function was supplied; if given only an IT, it returns the new transform module. The latter can then be used in other TRANSFORM calls to provide a base TT.

The full syntax of a call to TRANSFORM depends on the particular transformations used; one version will be described when the sample transformation system to be used here is described.

The way in which user code can access and use TT parameters will now be discussed.

For the purpose of examples, we will assume here the use of a simple set of transformations: translation, rotation, and a single scale factor applied to both X and Y coordinates. Given these transformations, a TT consists of four parameters: an X translation, a Y translation, a scale, and a rotation about the center of the display device's coordinate system. These parameters are the values of identifiers bound in a transform module's local environment; in particular, the identifiers are CEN, SCL, and ROT. CEN is bound to a position-valued output whose X and Y elements are the X and Y

translation; SCL is bound to an output containing a real (floating-point) ratio between distances under the TT and distances on the display device screen; and ROT is bound to an output containing a real (floating-point) rotation in radians about the origin of the display device.

By the previously presented definition of the association of a TT with a module, simply making the above ATOMs "EXTERNAL" identifiers obtains the parameters appropriate to any given picture module. As an example, a LINE module appears below which does its own translation, scaling, and rotation, presenting coordinates to the display file which are ready for immediate display; it should be noted that the AS-NEEDED daemon runs if either endpoint changes or if any of the transformation parameters changes:

```
(DEFPIC LINE (P1 P2 "AUX" LINEID
              "EXTERNAL" CEN SCL ROT)
   (SETQ LINEID
         (MAKE-LINE-ENTRY (TR ,P1 ,CEN ,SCL ,ROT)
                          (TR ,P2 ,CEN ,SCL ,ROT)))
   (AS-NEEDED
         (CHANGE-LINE-ENTRY LINEID
                            (TR ,P1 ,CEN ,SCL ,ROT)
                            (TR ,P2 ,CEN ,SCL ,ROT)))
   (ONC DELETE () (DELETE-LINE-ENTRY LINEID)) )
```

The function TR, which actually does the transformation, is defined as follows:

```
(DEFINE TR (P C S R)
   (+ C (* (POS (+ (* (X P) (COS R)) (* (Y P) (SIN R)))
                (+ (* (Y P) (COS R)) (* (X P) (- (SIN R)))))
           S)))
```

The above code is primarily for illustrative purposes; it may also be considered a definition of the way the assumed transformations affect the visible image. However, it does not parallel a real implementation, both because it uselessly recomputes trigonometric functions and because it does not lend itself to utilizing the available hardware.

A more reasonable implementation method is presented below; before launching that discussion, however, note must be made of the fact that especially if hardware support is used, the most convenient internal formats of TT parameters will undoubtedly not be formats reasonable for

user perusal. For example, a more internally convenient and efficiently usable form for the parameters assumed above is a 3X3 or 3X2 matrix in which the scale factor only exists multiplied by rotation parameters [New2]. If the user is to obtain access to parameters which are more readily usable in his code, the system must create daemons to translate internal TT parameters into the appropriate forms; various subterfuges can be used to assure that the minimum number of such daemons are created.

A more reasonable implementation of transformations can be performed as follows:

The transform modules described above are still used, and still contain several daemons and outputs which will be described below.

The display file is segmented, with one segment associated with each transform module. A segment is created when its corresponding transform module is created, and deleted when the transform module is deleted; the latter could be done by a DELETE daemon in the transform module and the former is readily done by TRANSFORM itself. Appropriate pointers to each display file segment are kept in the local environment of a segment's associated transform module.

Each display file segment contains data for drawing the visible images created under the TT associated with the segment's transform module. Each segment also contains display hardware commands to do whatever part of the transformation the hardware can manage. If this includes transformation concatenation, a segment may also include "display subroutine jumps" to the segments of all transform modules using as their base TT the segment's TT; this generally facilitates concatenation.

Aside from the display file segment, the elements of a transform module -- daemons and further auxiliary storage -- depend on the particular capabilities of the hardware. Three cases of a fairly general nature will be discussed here; other situations can be described

in terms of these three. The cases are: (1) the hardware is eminently capable of performing all the transformations desired as well as concatenating transformations; (2) the hardware can perform all transformations, but cannot do concatenation; (3) the hardware cannot perform concatenation and can perform only some of the transformations. The case where the hardware is incapable of any transformations seldom occurs, and at any rate is a simple extension of the third case. Each of these three cases will now be described.

Even if the hardware is omnipotent, one daemon is needed. This daemon watches all the IT outputs specified by user code, and when any of these change it formats the IT data properly and deposits it in the appropriate place(s) in the display file segment. In this case, LINE, STATLINE, and similar "display" modules format their own data and place them directly in the display file.

If the hardware can transform but not concatenate, at least one output for each TT will be required to propagate changes in TTs. This will be called the change output; it is changed -- perhaps toggled from true to false and back again -- whenever its associated TT is changed. An entire TT need not be passed via the change output since TT data can be accessed without literally watching it in an output. As was the case with very powerful hardware, here there is one daemon; in this case it watchs both the IT outputs and the change output associated with the base TT being used. This daemon also specifies the change output of its own TT. When either the IT or the base TT changes, the daemon (1) concatenates the IT and the base TT; (2) for the benefit of others using this TT as a base TT, updates the resultant TT parameters held in its own transform module's local environment; (3) formats the resultant TT appropriately and places it in its display file segment; and (4) OUCHes the change output to let others know that this TT has changed. "Display" -- e.g., LINE -- modules' daemons still place their data directly in the appropriate display file segment.

If the hardware is not only incapable of doing concatenation but

also cannot perform at least some of the transformations, a buffer separate from the actual display file segment is needed to hold untransformed display data produced by "display" modules. This buffer is best considered as an array of outputs, each OUCHed by an associated "display" module to change its contents; this need not literally be the case. This time two daemons are needed. The first is a "concatenation" daemon similar to the one used in the preceding case. It watches the IT outputs and base TT change output, does the concatenation, places the resultant TT in the local environment, OUCHes its own change output, and possibly formats that portion of the TT of which the display is capable; it does not, however, alter the display file segment. The second daemon, a "transformation" daemon, watches its own TT's change output as well as all the "pseudo-outputs" in the untransformed buffer. This daemon applies to the buffer entries that part of the TT which is beyond the capabilities of the hardware's capabilities, formats the results of that operation, and places the formatted results in the display file segment. If the "transformation" daemon is running because the TT has changed, it should also deposit in the display buffer the formatted version of those transformations within the hardware's abilities, performing the formatting itself if the "concatenation" daemon has not done so. The untransformed buffer must be transformed <u>en masse</u> if the TT has changed; if it has not -- i.e., the "transformation" daemon is running because some LINE-specified "pseudo-outputs" have changed -- then only the changed entries need be transformed. These entries can perhaps be selected on the basis of a "change bit" associated with each entry.

The reason why two daemons are necessary in the last case, rather than just using a single daemon to do everything, derives from the fact that the "transformation" daemon effectively depends on outputs specified by LINE modules and hence runs after all those LINE modules have run. This is not generally possible for the "concatenation" daemon, since the LINE change daemons may be web ancestors of the

"concatenation" daemon; this may occur if the user makes active use of TT outputs as described earlier. Making the transformation daemon separate avoids potential data web circularity which, aside from being illegal in DALI as so far presented, could lead to a deadlock.

Now the particular transformations chosen for use here, along with a syntax for TRANSFORM, will be described. In order to ease the description, some terminology and some characteristics of the transformations to be used will first be described.

The coordinate system used by modules associated with a TT A will be referred to as A's coordinate system.

Since the transformations chosen for use here include clipping, it will be convenient to define a resultant TT A in terms of a mapping from an area in A's coordinate system to an area in the coordinate system of A's base TT; these areas will be called the master space and the instance space, respectively. The master space is the space in which a module considers itself operating; presumably the module will draw something there. The instance space is the place where a module's caller puts the objects which the module draws. The master-instance space relationship simultaneously specifies translation, scaling in both X and Y, and, as done here, rotation. Other methods could be used, perhaps as optional equivalents to the space relationships chosen.

It should be noted that due to concatenation of clipping ("boxing"), a TT actually defines a mapping from some sub-area of the master space, commonly called a window, to some sub-area of the display screen, commonly called a viewport; however, a TT is locally defined in terms of master and instance spaces.

The master and instance spaces are rectangular areas. Given rotation, however, clipping against an arbitrary polygon is necessary; this is pointed out in [New2], and can be handled by the techniques mentioned in [Sut2]. However, if polygonal clipping is actually available, there is little reason to restrict the master and instance

spaces to being rectangles; they could be arbitrary polygons which are congruent under rotation and separate X and Y coordinate scalings. While a system utilizing master and instance spaces which are arbitrary polygons could be defined, it will not be done here; doing so would increase the complexity of the situation to no good explicatory end.

Initially it will be assumed that all master spaces are identical, ranging from -1000 to 1000 in both X and Y. These are arbitrary default values; the manner in which they can be changed will be described after discussion of the call to TRANSFORM, which follows.

An application of TRANSFORM has the following form:

(TRANSFORM -specs- picfun -args-)

The picfun argument is a picture function, and -args- are the arguments to be applied to picfun in creating a new picture module; this module will be the son of a new transform module, as was discussed earlier in this section. Both picfun and -args- are optional.

The -specs- arguments specify the resultant TT's base TT and the instance space portion of the IT. Since there are several such arguments, and all are optional, some simple syntax will be used to make calls to TRANSFORM legible; this syntax is described below.

The -specs- consist of groups of objects, each starting with a designator describing one or two supplied arguments; the supplied arguments immediately follow their designator and terminate the group. The order in which groups appear is not significant. There are five designators: "USING", "CENTER", "HALFSIZE", "ROTATION", and "CLIP". The arguments which follow the designators, the way the arguments are used, and their default values will now be described.

"USING" is followed by a picture module; the TT associated with this module is the base TT to be used. The default is the owner of the daemon applying TRANSFORM.

The remaining designators specify the instance space portion of an IT. It should be noted that they <u>all</u> refer to the base TT's coordinate system. This is a modularity measure designed to decrease the amount

that needs to be known about a module, in particular the way its coordinate system is arranged.

"CENTER" is followed by a position-valued output; the position is the location in the base TT's coordinate system of the center of the instance space. The default is the center of the base TT's master space.

"HALFSIZE" is also followed by a position-valued output; the position's X and Y values are one-half the width, and one-half the height, of the instance space; if the instance space is upright, the "HALFSIZE" position is the position of the upper-right corner of instance space relative to the center of the instance space. The default values are one-half the height and width of the base TT's master space.

Use of only "CENTER" and "HALFSIZE" can result in clipping, since the instance space they specify need not lie entirely within the base TT's master space.

"CLIP" is followed by two arguments, both position-valued outputs. The first is the center, and the second one-half the height and width as in "HALFSIZE", of an area in the base TT's coordinate system; this area is called the clipping area. Only within the clipping area are objects and parts of objects visible when drawn in the resultant TT. Usually, this area will overlap with the space designated by "CENTER" and "HALFSIZE", since only within the overlapping area can anything appear. The defaults are the "CENTER" and "HALFSIZE" arguments.

The clipping area has no effect on the instance space; its purpose is to allow a user to select for display a particular piece of the image produced by a module. It will often be useful to specify the "CLIP" arguments by means of a daemon watching the "CENTER" and "HALFSIZE" arguments; for example, doing so is the natural way to make only the lower-left quadrant of the created image visible.

"ROTATION" is followed by an output containing a real (floating point) number; this specifies the rotation in radians. Rotation poses a

problem: exactly what is rotated, and about what position? Here we chose to rotate the "CENTER"/"HALFSIZE" and clipping areas about the "CENTER" argument. The area resulting from rotating the "CENTER"/"HALFSIZE" area in this manner is the instance space.

Doing rotation in the manner described above produces tilted and arbitrarily polygonal areas against which clipping must be done. As noted previously, this can be done [Sut2], but it is significantly less efficient than clipping against an upright rectangle. However, the author knows of no way to incorporate both general clipping and general rotation in the same transformation system without clipping in this fashion.

As an example of the use of TRANSFORM, here is a different way to create the variable-sided polygon of the preceding example:

```
(DEFPIC NGON (N C R "AUXO" DTHET UR)
    (CONTIN (OUCH DTHET (/ 6.2832 ,N)))
    (CONTIN (OUCH UR (POS ,R ,R)))
    (DODA N (LIST (NULLSPEC (OUTPUT (POS 1000 0)) 0)
        TRANSFORM "CENTER" C "HALFSIZE" UR
            CORNER (PREOUT 1) (PREOUT 2)) )

(DEFPIC CORNER (PP THET "EXTERNAL" DTHET "OUT" NP NTHET)
    (CONTIN (OUCH NTHET (+ ,THET ,DTHET))
            (OUCH NP (* 1000 (POS (SIN ,NTHET)
                                  (COS ,NTHET)))) )
    (LINE PP NP) )
```

Here we have used the "CENTER" and "HALFSIZE" mechanisms to do the requisite scaling and translation, eliminating CORNER's explicit dependence on C and R, and instead hiding it in a transform module.

The method of specifying the bounds of a master space must still be discussed. This operation is performed by using particular reserved words, i.e., reserved ATOMs, in the argument list of a picture function; these ATOMs are CENTER, HALFSIZE, and ROTATION. They may be used as argument, "EXTERNAL", "AUXO", or "OUT" identifiers (section 3.6); but in any case, they must be bound to outputs. The values of CENTER, HALFSIZE, and ROTATION respectively specify the master space's center, halfsize, and rotation about the CENTER position in the manner of the

"CENTER" "HALFSIZE", and "ROTATION" arguments to TRANSFORM. The values of the outputs bound to these ATOMs are specially initialized if they appear as "OUT" or "AUXO" identifiers: CENTER receives the position (0,0), HALFSIZE receives the position (1000,1000), and ROTATION receives 0. If any of these three ATOMs appear as arguments or as "EXTERNAL" identifiers, they are bound in the normal fashion; but CENTER and HALFSIZE must be bound to position-valued outputs, and ROTATION must be bound to an output containing a real (floating-point) number.

A picture function containing any of these ATOMs in its argument list is treated in a special way when applied to arguments. If it is applied as part of an application of TRANSFORM, the outputs to which they are bound are used in the created transform module in defining the resultant TT. If such a picture function is not applied using TRANSFORM, it is treated as if it were applied using a call to TRANSFORM in which all the other arguments received default values.

As an example of how master space manipulation can be used, here is yet another recoding of NGON:

```
(DEFPIC NGON (N C R "AUXO" DTHET UR EP CP)
    (CONTIN (OUCH DTHET (/ 6.2832 ,N)))
    (CONTIN (OUCH UR (POS ,R ,R)))
    (CONTIN (OUCH EP (POS (SIN ,DTHET) (COS ,DTHET))))
    (OUCH CP (POS 1 0))
    (NULLSPEC CP)
    (DODA N (LIST (NULLSPEC (OUTPUT 0)))
       TRANSFORM "CENTER" C "HALFSIZE" UR
          CORNER (PREOUT 1)))

(DEFPIC CORNER (ROTATE "EXTERNAL" DTHET EP CP
                       "AUXO" HALFSIZE "OUT" NR)
    (OUCH HALFSIZE (POS 1 1))
    (NULLSPEC HALFSIZE)
    (CONTIN (OUCH NR (+ ,DTHET ,ROTATE)) )
    (LINE CP EP) )
```

Here each CORNER operates in a constant space ranging from -1 to 1 in X and Y, and is mapped into a square space of side 2R centered on C; this provides scaling and translation as in the last example. Furthermore, each successive CORNER master space is rotated by DTHET from its predecessor by making ROTATE an input and appropriately controlling the NR output. This means that the numeric positional values of every line's endpoints are the same. That is taken advantage

of by sharing these endpoints; one, CP, is constant at (POS 1 0), and the other, EP, varies depending on N. Note that NULLSPEC is used to allow LINE to use an otherwise constant output. Making the CORNER master space of halfsize (1,1) simply removes the explicit multiplicative factor from the computation of EP, and, again, hides it in a transform module.

There is a further complication to transformations which must be dealt with in DALI.

In the last NGON example, we utilized the fact that particular numeric coordinates, valid in one space, had a meaningful interpretation in another. This is not always the case. In general, suppose a module A is operating in a space whose total transformation is represented by a 3x3 matrix TA. Thus the display coordinates of a position (X,Y) in A's coordinate system are obtainable from

$$[X\ Y\ 1] * TA,$$

where * indicates matrix multiplication. Further suppose that said module uses that position as the value of an output, and that another module, B, operating under a different TT TB, has that output as an input. Clearly, A's literal X and Y values generally represent sheer gibberish to B. The intelligible versions of them are some (X',Y') expressed in B's coordinate system. These are defined by

$$[X\ Y\ 1] * TA * TB' = [X'\ Y'\ 1]$$

where TB' is the inverse of TB. TB' always exists if TB in fact represents only concatenated translations, rotations, and scalings; this condition also guarantees the unit third element of the result.

It is a fact that sophisticated users of graphics systems with fairly general transformational capabilities quite soon run into the necessity of creating TA*TB' transformations. This happens, for instance, whenever an input device is used for drawing inside a transformed section of a display.

Since an output contains pointers to both its owner and all its

dependent modules, we can always find TA and all the TB's, and thus it is always possible to provide meaningful coordinates to any module B. However, we are dealing here with a potentially enormous amount of computation, and it behooves us to treat it as a special case: even display hardware which glibly performs transformation concatenations isn't going to invert TB for us as a regular part of the display refresh cycle, if at all. Therefore we provide a special built-in picture function RESPACE:

(RESPACE outp)

where outp is a position-valued output. The picture module resulting from applying RESPACE has an output which is the position corresponding to outp's value under the TT in which RESPACE is applied. Various subterfuges can be resorted to for avoiding unnecessary inversions of TB when many similar RESPACEs are done.

4.3 Deletion

The effect of deletion, performed by the function DELETE, is to cause an object to cease to exist. Deletion can be performed on picture modules, daemons, outputs, and several other objects which will be mentioned. Deletion serves two purposes: primarily, it removes from the picture definition all effects of the object deleted; with that done, the storage occupied by the object can be reclaimed and so it is, as the secondary purpose.

The subject of deletion invokes a serious meta-question: Why have explicit deletion rather than retentive storage management -- i.e., garbage collection? This question will be discussed before the problems and mechanics of deletion are addressed.

It must initially be stated that the use of explicit deletion is

not motivated by questions of efficiency. Deletion in fact has two disadvantages which offset any greater efficiency that it may have:

First, deletion significantly increases the size and complexity of nearly every DALI object. This occurs because whenever an object A makes use of an object B -- i.e., A contains a pointer to B -- there must then be some path by which B can refer back to A so that if B is deleted, A can be deleted or appropriately modified. For example, a module needs a pointer to its containment tree father in order to resolve "EXTERNAL" references; hence the father must point back to all his sons so that when the father is deleted, the sons can also be deleted. Whether this is accomplished with two-way pointers, rings as in SKETCHPAD [Sut1], or some other means, it represents an overhead which does not exist when garbage collection is used.

Second, deletion presents modularity problems: A module may contain, in its local environment, pointers to an arbitrarily large and complex data structure constructed by the user-written code of that module's daemons. If the module is deleted, at least part of that data structure must be deleted -- and there is no way for the deletion procedure to know what part. This problem is solved in DALI with a device called a "deletion p-closure", which is described later in this section; it is not solution of which the author is fond, as it is imperfect and embodies significant overhead.

The reason why deletion is used derives from the fact that the internal DALI data structures contain an inherent loop with one particularly troublesome pointer: the pointer from an output to a daemon which is to be queued when the output is OUCHed. The full argument establishing the reason why this loop apparently makes garbage collection impossible necessarily involves the construction of a "gedanken" garbage-collected DALI. Because of its size, this construction has been relegated to Appendix 3; a relatively abstract summary of the central part of the argument follows.

An output contains references to the daemons watching it so that

those daemons can be queued when the output is OUCHed. This could keep a daemon -- and its owner -- from being reclaimed unless all its watched outputs were reclaimed, leading to the retention of much useless storage and the running of many useless daemons. Furthermore, it is unreasonable to require the user to delete these references by explicit programming, since they are constructed by the DALI system itself. Instead, the garbage collector can ignore these references, collect the useless daemons, and appropriately modify the sets of pointers contained in outputs. Unfortunately, the garbage collector runs at infrequent intervals, and until it runs and changes an output's set of "watching" daemons, the useless daemons will still be queued and run.

This is an untenable situation. If daemons continue to run for an indefinite period after they "should" not exist, it is very likely that they will fail catastrophically because output values are in what is, for them, an inconsistent configuration. So garbage collection of DALI objects must, regretfully, not be used.

It must be noted, however, that the above discussion says nothing about the possibility of garbage collecting user storage, i.e., letting the user construct "free" objects and garbage collecting them when they are no longer used. This is quite possible, and will be a convenience; however, given deletion p-closures, it is not a necessity.

Now the manner in which deletion is accomplished will be presented.

Deletion in DALI is akin to the recursive deletion of SKETCHPAD [Sut1], in that when an object is deleted, all objects which "depend on" its existence are also deleted.

The dependence of an object O on a deletable object D can take one or both of two forms:

(1) O can be part of D. For example, the daemons and outputs owned by a module are part of that module, as are the module's sons. This will be called <u>containment dependence</u>, and is expressed by the containment tree and ownership relations (section 3.5).

(2) The correct operation of O can require the existence of D. For example, a daemon's accessing the value of a local environment identifier is erroneous if that value is an object which has been deleted. This will be called operational dependence.

In the case of containment dependence, deletion of D requires deletion of O for logical consistency; this is therefore done in DALI. In the case of operational dependence, it is often possible and desirable to modify the dependent object O so as to permit its continued operation. For example, rather than being deleted when one of its watched outputs is deleted, a daemon might be kept in operation by substituting some other "default" output for the deleted one. However, the nature of the required modification depends upon the way the objects are being used; no single uniform procedure will always be correct. DALI therefore provides a daemonic mechanism called a deletion p-closure, to be discussed later, by which the user can specify his own desired modifications under deletion. If a deletion p-closure is not used, or if used does not remove detectable dependence, the dependent object is deleted.

In the example of operational dependence in the definition above, deletion of an object assigned in a local environment causes deletion of the module containing that local environment, since the latter is no longer well-defined. Thus, for example, deletion of either endpoint of a LINE causes the LINE to be deleted.

It should be noted that maintenance of correctness by dependence deletion must be done by the DALI system, because it cannot be done by the user without sacrificing the modularity afforded by DALI: a module does not -- and should not have to -- know what other modules depend on its outputs. The principle mechanism for maintaining consistency through recursive deletion of operationally dependent objects is the set of dependents; this will be discussed in detail further below.

It is often the case that when an object is deleted, there are

certain local "cleaning up" operations that must be done. For example, since a LINE module creates and controls an entry in the hardware display file, that entry must be deleted if the LINE is deleted. A further example concerns a module which has created, and stored in its local environment, a list composed of storage cells from a (non-garbage-collected) heap. When the module is deleted, it must be given a chance to return that list to "free" storage.

Such actions must be user-definable on a local, modular basis. To permit this, DALI uses a device called a deletion p-closure (DPC). The term "p-closure" is used to indicate an analogy with functional closures; the "p-" indicates that the function is closed with respect to a picture module's local environment.

A DPC has three elements:

(1) an owner, which is a picture module

(2) an undertaker, which is a function of 1 argument

(3) a corpse, which is a deletable object.

The owner is that picture function which was current when the DPC was created. The undertaker is a user-defined function whose returned value is ignored (a procedure).

The corpse is that object whose deletion the DPC "waits for".

Just before a DPC's corpse is deleted, the DPC's undertaker is applied to the corpse as if the undertaker were a daemon of its owner. This is done as part of the deletion of the corpse. When the undertaker finishes, the corpse is interred -- i.e., it is "really" deleted and its storage is reclaimed. If a corpse has multiple DPCs, the order in which they are run is undefined.

Thus a DPC acts very much like a daemon dependent on a mythical output which is OUCHed just before deletion of its corpse.

A DPC is created by execution of

(ONDELETION corpse undertaker)

This application returns the newly-created DPC.

The (ONC DELETE ---) construction presented previously in the LINE module example is syntactic sugaring for

```
(ONDELETION (CURRENT-MODULE)
            (LAMBDA (MODULE) ---))
```

where (CURRENT-MODULE) returns the current module.

A deletion p-closure is, as its name suggests, a particular daemonic use of an M-DALI object called a p-closure, a functional closure with respect to a module's local environment. While the author realizes that a discussion of p-closures in the midst of a discussion of deletion is scarcely appropriate, he can think of no other place where the motivation for them springs from the material under discussion. So:

(P-CLOSURE func mod)

where func is a function and mod is a module, creates and returns a closure of func with respect to mod. mod is optional and defaults to the owner of the daemon currently executing.

A p-closure is applied to arguments as if it were the func argument of P-CLOSURE. However, in the execution of func's body, the local environment of mod is used in place of whatever local environment existed when the p-closure application was done. A realistic example using p-closures appears in section 5.9.

With the addition of p-closures and deletion p-closures, all the deletable objects of M-DALI have been introduced: outputs, daemons, picture modules, p-closures, and deletion p-closures. S-DALI will add two more: sequences and scheduled p-closed actions; these two are described in sections 6.3 and 6.4.

Each of these deletable objects has, by virtue of being deletable, three elements which are used only in deletion and have previously been unmentioned:

(1) a mark bit

(2) a set of deletion p-closures (DPCs)

(3) a set of dependents.

The mark bit is needed to avoid infinite recursion during deletion when circular data webs are used.

The purpose of DPCs has been discussed. A DPC is a member of this set if and only if the object containing this set is the DPC's corpse.

The set of dependents contains picture modules: every module M with a deletable object D bound to an identifier in M's local environment is in D's set of dependents. Its use is discussed further below.

The set of dependents is a mechanism for enforcing consistency by making it difficult for a user to reference storage which has already been reclaimed. The enforcement is performed by deleting all the dependents of an object when the object is deleted. This operation is called implicit dependence deletion; it causes, for example, the deletion of a LINE module when either of its endpoints is deleted.

It is clear that maintenance of the set of dependents can be fairly expensive, as it must be done at run time by the operations of binding and assignment of local environment identifiers. Since the function of this set can be performed by appropriate DPCs, which in many cases will be more efficient, it is appropriate to allow a confident user to specify a mode of operation or compilation in which assignment does not include this overhead. Alternatively, only initial binding of inputs could add to the set of dependents. The situation is somewhat similar to that of run-time array bounds checking.

Furthermore, implicit dependence deletion, as defined, is an imperfect protection mechanism: the user can always assign a structure containing a deletable object to a local environment identifier, thereby bypassing the mechanism. This case could be handled by adding a dependent structure set, kept updated by structure creation and modification operations. Deletion of an object would then also entail replacing the object in the structure with an "obviously incorrect" value, such as 0, thereby allowing validity tests to be made whenever a free object is used. Since, however, deletion of objects in structures will usually require a "cleanup" operation which alters the structure, DPCs will normally be used if such deletion is to be feared; hence the dependent structure mechanism has not been included.

Why only identifiers in local environments need be considered, as opposed to temporary environment identifiers also, follows from the nature of the deletion process and will be discussed.

Now the actual process of deletion will be described.

DALI utilizes delayed deletion. On execution in a daemon or picture function body of

(DELETE obj)

where obj is a deletable object, obj is not immediately destroyed. Instead, obj is inserted into a global deletion set. Then, after termination of daemon body execution, the members of the deletion set are interred, i.e., "really" deleted and removed from the deletion set. The driving program is an exception to this rule, as explained further below.

The purpose of delayed deletion is to wait until the total DALI environment is in a known state. When interment occurs, the only temporary environment in existence is that of the driving program, thereby limiting implied dependencies among "real" picture modules to the local environments. This is considered adequate because in DALI, as opposed to most display systems, the information about the picture held in the driving program's local environment can be kept fairly simple.

Whenever the total environment is in the described minimal state, interment is performed on all objects currently in the deletion set. This minimal state occurs often and is easily detected: It occurs whenever the scheduler has just finished running an entry in the daemon queue. It also continuously occurs while the driving program is running; hence, deletion when done by the driving program is not delayed.

A further purpose in delaying deletion is to allow objects time to get out from under the hammer of implied dependence deletion. After all, it is quite possible that DELETE's argument was the value of a local environment identifier.

Interment will be described in a ouasi-programmatic form.

A uniform prelude of three steps begins the interment process for all deletable objects. For an object O, these are:

(1) Test the mark bit of O; if it is 1, return immediately. Otherwise set it to 1 and continue.

(2) Run and inter all of O's deletion p-closures.

(3) Inter all of O's dependents.

In addition, a uniform coda ends the interment process: O's storage is returned t he "free" storage pool.

The remainder of the steps differ for each type of object, and are detailed below. The general rule they follow is: first inter any modules which need it, then any outputs, then any daemons, and only then perform the local unlinking operations required for the object.

Interment of a picture module P:

(4) inter P's sons

(5) for each output OU owned by P,

(5.1) run all deletion p-closures of OU

(5.2) inter all modules in OU's set of dependent modules

(5.3) inter all daemons watching OU

(5.4) if OU has a specifier S, remove OU from S

(6) inter the daemons, deletion p-closures, and p-closures owned by P

(7) for each deletable object DO bound to an identifier in P's local environment, remove P from DO's set of dependents.

(8) remove P from its father

Interment of a daemon D:

(4) make all of D's specified outputs unspecified

(5) for each watched output OU of D, remove D from OU

(6) if D is queued, remove it from the daemon queue.

(7) remove D from its owner

Interment of a deletion p-closure DPC:

(4) remove DPC from its owner.

This completes the discussion of deletion in M-DALI. Deletion in S-DALI is identical, barring the existence of more types of deletable objects.

4.4 Further Daemonology and Data Web Change

The basic mechanisms behind the operation of daemons and outputs in DALI have been presented. Here some simple but powerful and necessary extensions of those mechanisms are presented, leading into structural change to the data web.

The first extension is a means by which a daemon can determine which of its watched outputs actually changed. This is done with a new type of daemon, called a named-change daemon, created by

(NAMEDONC atm cndtn (-specs-) -body-) .

cndtn, (-specs-), and -body- are the daemon's condition, specified outputs, and body as usual. atm is an ATOM which, just before the daemon's body is executed, is bound in the temporary environment to a list of all the watched outputs which have changed. The order of the outputs in the list is undefined. That list, called the changed-outputs list, is held while the daemon is queued in an element of this type of daemon. After running the daemon, the changed-outputs list is emptied.

NAMEDONS, with semantics similar to ONS, also exists; when the daemon is run at its creation, the changed-outputs list is NIL.

An example:

```
(DEFPIC MERGE (O1 O2 "OUT" EITHER)
     (OUCH EITHER ,O1)                    ;Arbitrary initialization.
     (NAMEDONC HIM (VAL O1 O2) (EITHER)
        (OUCH EITHER ,(CAR HIM)) ))
```

If either O1 or O2 changes, but not both, MERGE's output mirrors the most recent value of either. If both change "simultaneously", the output value will be that of one of them, but which one is undefined.

Named-change daemons are a powerful extension; they effectively convert daemons from procedures with no arguments to procedures with arguments. Moreover, given named-change daemons, there are several new, useful capabilities that can be added to the daemonology of DALI. The rest of this section describes such additions.

The first new capability is the ability to make a daemon depend on or specify a set (list) of outputs without mentioning all of them by name. This is done by placing the fragment

"MEMBERS OF" lst ,

where lst evaluates to a list, in a daemon's condition or specified output list in place of a single output. For example, if OUTL is bound to the list (o1 o2 o3) and PQ is bound to o4, then both of

(NAMEDONC HIM (VAL PQ "MEMBERS OF" OUTL) . . .)

(NAMEDONC HIM (VAL "MEMBERS OF" OUTL PQ) . . .)

have the same effect, causing the daemon to watch o1, o2, o3, and o4.

Similarly,

(ONC (VAL . . .) (PQ "MEMBERS OF" OUTL) . . .)

causes the daemon to specify o1, o2, o3 and o4.

An example:

```
(DEFPIC SWITCH (SELECTOR LST "AUX" SELECTED
                "AUXO" DING "OUT" SELVAL)
    (OUCH DING T)
    (ONS (VAL SELECTOR) (DING)
       (SETQ SELECTED (NTH LST ,SELECTOR))
       (OUCH DING (NOT ,DING)) )
    (OUCH SELVAL ,SELECTED)
    (NAMEDONC WHO (VAL DING "MEMBERS OF" LST) (SELVAL)
       (COND ((OR (MEMQ DING WHO) (MEMQ SELECTED WHO))
                (OUCH SELVAL ,SELECTED)) )) )
```

SELECTOR is an output whose value is an integer, and LST is a list of outputs. The LISP function MEMQ is a boolean, returning <u>true</u> if and only if its first argument is a member of its second argument. SWITCH makes its output "follow" the value of the SELECTOR<u>th</u> element of LST; thus it corresponds to a multi-throw electrical switch. A "multi-pole switch" could be constructed by making several SWITCH modules with disjoint LSTs share a common SELECTOR. DING is the sound of a bell

waking up the second daemon when the SWITCH is "thrown", thus assuring that SELVAL attains the correct value of the selected output whether or not the latter is OUCHed along with the selector. The above implementation of SWITCH is rather inefficient, since the second daemon is run when any output in LST changes, not just the currently selected one. A better implementation will be presented below.

The next extension provides for varying a daemon's watched and specified outputs dynamically.

A daemon dem can be made to specify a set of outputs -outs-by execution of

(SPECIFIES dem -outs-)

where -outs- is a number of outputs none of which has a specifier. Similarly,

(WATCHES dem -outs-)

makes the daemon dem watch the outputs -outs-.

The inverses of SPECIFIES and WATCHES are, respectively

(UNSPECIFY -outs-)

(UNWATCH dem -outs-) .

A restriction: UNWATCH may only be applied to named-change daemons. The reason is provision for this feature: the affected output(s) are removed from the affected daemon's changed-output list if they are on it; if this empties the list, the daemon is removed from the daemon queue. Thus the sequence

```
(WATCHES dem out)
(OUCH out val)
(UNWATCH dem out)
```

does not cause dem to run.

As an example of the use of WATCHES and UNWATCH, here is the more efficient switch promised earlier:

```
(DEFPIC SWITCH (SELECTOR LST "AUX" SELECTED POLE
                "AUXO" DING "OUT" SELVAL)
     (OUCH DING T)
     (SETQ SELECTED (NTH LST ,SELECTOR))
     (OUCH SELVAL ,SELECTED)
     (ONC (VAL SELECTOR) (DING)
        (PROG (SEL)
             (SETQ SEL (NTH LST ,SELECTOR))
          ;If a new different output is selected,
          ;swap outputs for the POLE daemon.
             (COND ((NOT (EQ SELECTED SEL))
                    (UNWATCH POLE SELECTED)
                    (SETQ SELECTED SEL)
                    (WATCHES POLE SELECTED)
                    (OUCH DING (NOT ,DING)) )) )
     (SETQ POLE
        (NAMEDONC WHO (VAL DING) (SELVAL)
             (OUCH SELVAL ,SELECTED) ))
     (CAN-WATCH POLE LST)                    ;See following text.
     (WATCHES POLE SELECTED) ))
```

This is more efficient than the previous SWITCH in that the data-passing daemon, POLE, here (1) does not have to search a list to find out what to do, and (2) runs a minimum number of times: only when the selected output is OUCHed and when a different output is selected. The mysterious function CAN-WATCH is an efficiency measure discussed at the end of this section.

The functions SPECIFIES and WATCHES, while necessary, pose two problems if used in the unrestricted manner implied above:

First, it is no longer possible to guarantee a lack of cycles in the data web without explicitly inspecting it. In fact, arbitrary data web formations can be created with the functions in question. Perhaps fortunately, the second problem implies a test for web circularity.

Second, even if execution of SPECIFIES and WATCHES does not create circularities, it may necessitate recomputation of the priorities of daemons in a swath of the data web. This arises with (WATCHES dem out) if the specifier of out has a higher priority than dem, and in (SPECIFIES dem out) if a daemon watching out has a lower priority than dem. In either case, existing priorities of both the daemons mentioned and their web descendants are no longer guaranteed to result in a correct queueing order, and must be recomputed in a recursive fashion.

This recursion must contain a test for web circularity; otherwise, inadvertently created circularity would lead to infinite recursion. The "mark bit" needed for deletion will be used for this test.

The recursive procedure RE-PRIORITIZE for recomputing priorities is described below. It is straightforward, but can be expensive; in the worst case, the time needed rises exponentially with the number of daemons in the data web. Applied to a daemon DEM, RE-PRIORITIZE does the following:

(1) if DEM is marked, call an error routine to announce circularity.

(2) let a variable P be 1 + (the maximum priority of DEM's fathers).

(3) if DEM's priority is greater than or equal to P, return.

(4) set DEM's priority to P.

(5) mark DEM.

(6) recursively call RE-PRIORITIZE on all of DEM's web sons.

(7) unmark DEM and return.

RE-PRIORITIZE is not a user-callable function; it is automatically applied by DALI to a daemon as part of (WATCHES dem -outs-) and to all the watchers of all the -outs- as part of (SPECIFIES dem -outs-). UNWATCH and UNSPECIFY do not require an analogous procedure, since it is unnecessary to ever decrease the priority of a daemon; furthermore, they cannot cause circularity.

The application of RE-PRIORITIZE is actually delayed, like deletion, until the currently executing daemon has terminated. This requires that WATCHES and SPECIFIES put the daemons they have polluted onto a global list, if they are not already there, so that RE-PRIORITIZE can find those daemons when daemon body execution has terminated. This delay avoids redundant applications of RE-PRIORITIZE when, for example, WATCHES is applied more than once to the same daemon by the same daemon.

The cost of RE-PRIORITIZING can be significantly reduced if the set of outputs which a daemon can watch is known. Given that set, the

daemon can be given a priority greater than all its possible web ancestors and then RE-PRIORITIZED, thereby causing immediate exit from all future RE-PRIORITIZEs at step (3). This efficiency measure is performed by the function CAN-WATCH, called as

(CAN-WATCH dem outl)

where outl is a list of the outputs dem can watch. This was used in the efficient SWITCH above. CAN-WATCH places no limitation on the outputs dem can be made to watch; it simply assures that any output in outl can efficiently be watched.

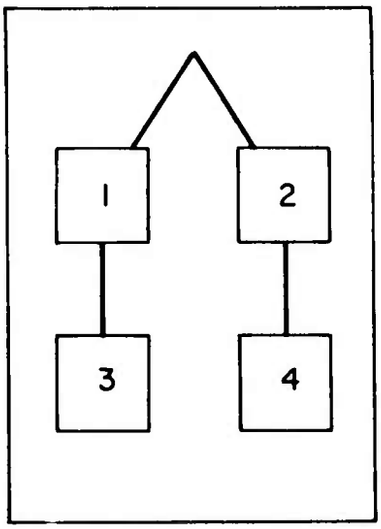
4.5 A Large Example: The Incredible Plastic Tree

A fairly substantial example, hopefully illustrating the power of DALI, will now be presented. It is the display of an n-ary branching tree which grows without bound, displayed in an area whose width is limited. The action desired in this display, called the "Incredible Plastic Tree" (IPT), is illustrated in Fig. 4-1. The characteristics desired of the IPT will first be described, followed by a discussion of a DALI implementation and ending with the code itself.

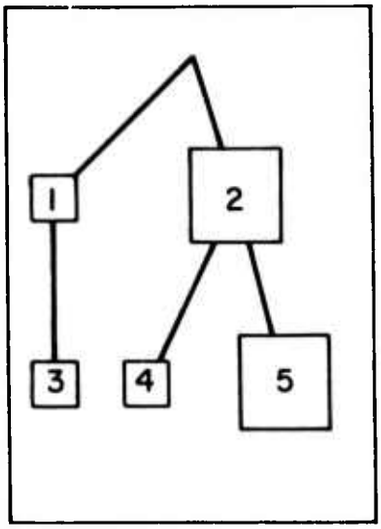
The IPT is a display of an n-ary tree which can grow boundlessly. The width of the display area available for showing the tree has an *a priori* limit, but the height is assumed to be adequate. The height could also be limited, but doing this would increase the complexity of the example without increasing its utility as an example of DALI "in action". No node on the tree is ever deleted; this restriction could also be removed, given the existence of deletion p-closures.

Each node of the IPT contains text which is to be displayed centered within a rectangular box. Son nodes on the IPT are connected

A



B



C

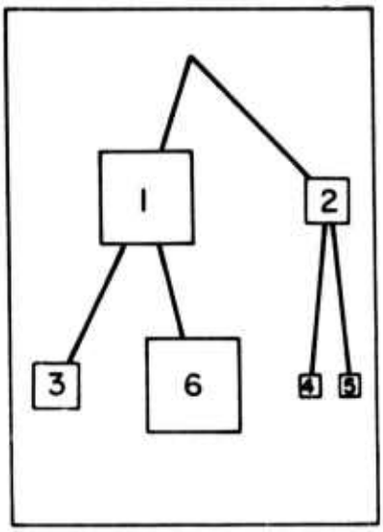


FIG. 4-1
WRITHINGS OF
THE IPT

to their father nodes by lines between the centers of their boxes which are "clipped" at the boxes' edges so as not to obscure the text.

Since the tree can grow boundlessly, it is apt to become too wide at any given level to allow all the nodes to remain at an "optimal" size for reading the text; so at least some nodes must shrink. Since it is most likely to be the case that a viewer is principally interested in the most recently added node, that node will be displayed at its "optimum" size. The algorithm used also has the nice effect of making ancestors of the most recently added node large, "optimum" at largest.

In general terms, what the IPT tries to achieve is this: given a quantity of (tree-)structured information too large for comprehensibly display in all its detail, the "currently important part" (the most recently added node) is shown in full detail, while less important parts are shown in less detail; at the same time, the structure in which the information is organized is always visible in at least schematic form (the whole tree is always displayed, perhaps with some nodes shrunk to dots), thus maintaining a display of the relationship between the "currently important part" and the other parts.

The action of the algorithm is illustrated in Fig. 4-1; from Fig. 4-1a, first node 5 is added to produce Fig. 4-1b; then node 6 is added to produce Fig. 4-1c.

As might be expected, the primary picture function used in implementing the IPT is recursive: it calls itself to create the display of a new son. This basic picture function is called NODE. Its task is to create sons and allocate area for display among its sons; the actual display of a node -- the box, the text, and the connecting lines -- is performed by DRAW-NODE, a picture function called by NODE.

NODE allocates area to its sons as follows: when one of its NODE sons requests "more area", either because it has just been newly created or because one of its sons has requested area, NODE passes the request on up to its NODE father via an OUCH. The top NODE starts the process

of re-allocating the available area among his sons, passing "area changes" back down the whole tree via OUCHes. The resultant data web structure is not circular because the NODE daemon requesting area from his father is disjoint from the daemon which accepts area changes and allocates them among his sons.

A fragment of a data subweb used for area control is illustrated in Fig. 4-2. In that fragment, the daemons labelled MA request More Area from their father, and the daemons marked AC meekly accept Area Changes from the father and pass them on to sons.

Each NODE provides its father with two outputs whose values are used to determine equitable allocation of area: the space the NODE wants, and the space it needs. The space wanted is the larger of (1) what it needs to display its own box and text "optimally", and (2) the sum of what its sons want. The space required for "optimal" display is obtained as a constant value from DRAW-NODE. The space needed is that which the particular son requesting more area needs; a newly created son "needs" its optimal area. While processing an area request, the More Area daemon records which son declared a "need" in a local environment identifier; this value is used by the Area Change daemon in allocating area.

The top NODE, on receiving a new "need", turns around and starts allocating area to its sons through outputs specifying the left and right boundaries of the area into which each son must fit both itself and all its sons. Area is allocated in proportion to the amount each son wants, except for that son, if any, who requested a new "need". That son gets the larger of his proportional share or his need.

New sons are generated as follows: In addition to arguments which are outputs specifying the area it gets and other data needed for display, each NODE is given an output called NEW-SON. NEW-SON initially contains the text string to be displayed; this is put in to display file during initialization. Whenever the NEW-SON output changes value, a new NODE son is to be created. The new value of NEW-SON is the NEW-SON

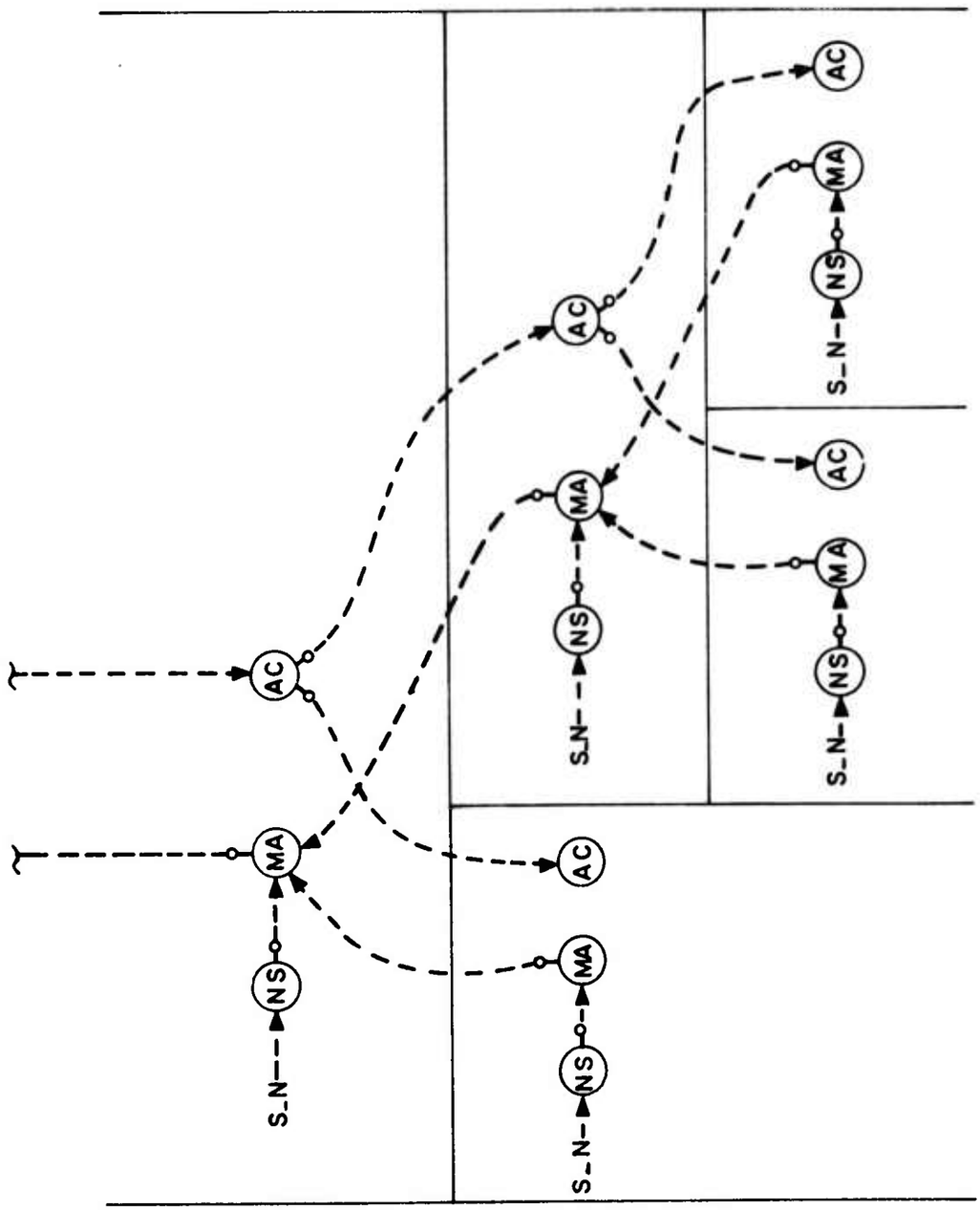


FIGURE 4-2 AREA CONTROL DATA WEB FOR THE IPT

output to be passed to the new NODE. Thus, the driving program just pumps new NEW-SON outputs through existing ones to create the display.

DRAW-NODE is much less interesting than NODE. Its arguments are: the text string to be displayed; the center and halfsize of its father NODE's box; and the right and left sides of the available area. The latter information is passed as the X and Y coordinates of a single position-valued output. DRAW-NODE uses the picture function BOXTXT to actually draw the text and box, using a coordinate transformation to scale the drawn box and text to fit in the needed area. The center and halfsize used in that scaling are outputs of DRAW-NODE so that NODE can pass them to new NODE sons.

The vertical location of the box is obtained by adding an "EXTERNAL" delta Y to the father's center, and the box is kept centered in the available area.

The optimal size of the box is obtained by examining the text string with TEXTHEIGHT and TEXTWIDTH, functions which return the optimal height and width for a text string; these will not be shown. The actual box is made larger than this area by a "fudge factor" BEAUTY, another "EXTERNAL"; this is to allow some empty margin. In addition, the box will not be shrunk until BEAUTY times the optimum width is no longer available; this similarly provides for space between horizontally adjacent boxes. Instead of a multiplicative factor, a constant margin width could have been used.

The fancy line between boxes is drawn in an interesting way by the picture function OUTERLINE. OUTERLINE uses a picture function CLIP, which takes four inputs and produces a module with three outputs. The inputs are two line endpoints, and positions specifying the center and halfsize of a clipping area. The outputs are: (1) A boolean flag which is true if and only if at least part of the line lies within the area; if this is false, the other outputs are not valid. In this application, it will always be true. (2) A position: the place where the first

endpoint winds up after clipping. (3) Another position: the place where the second endpoint winds up after clipping. CLIP will not be shown; [New2] gives an appropriate algorithm, which, while it does not preserve knowledge of which input position corresponds to which output position, could be easily modified to do so.

OUTERLINE uses two CLIPs, one for its node's box and one for its father's box, and hands each the endpoints of a line connecting the boxes' centers. Then one of the outputs of each CLIP is the desired endpoint on the edge of the box.

The code for NODE, DRAW-NODE, BOXTXT, and OUTERLINE, as well as several functions called by NODE, follows.

```
(DEFPIC NODE (NEW-SON I-GET                          ;Area which I GET.
              PA-CEN PA-HSZ  ;PApa NODE's CENter and HalfSiZe
             "OUT" I-WANT I-NEED      ;Tell pa what want and need.
             "AUX" SONS-WANT SONS-NEED MY-REQUEST      ;See below.
                   AREA-CHANGE MORE-AREA   ;The daemons. See text.
                   MY-CEN MY-HSZ   ;CENter and HalfSiZe of my box.
                   LOCAL-NEED           ;Optimal width for my box.
             "AUXO" DINGS DINGA)                      ;"Ding" outputs.
  (SETQ MY-REQUEST 0)       ;Flag -- number of son requesting or 0.
  (SETQ SONS-WANT ())                ;List of sons' I-WANT outputs.
  (SETQ SONS-NEED ())                ;List of sons' I-NEED outputs.
  (PROG (DRAW-IT)       ;Do actual drawing, save center & 1/2 size.
        (SETQ DRAW-IT (DRAW-NODE ,NEW-SON PA-CEN PA-HSZ I-GET))
        (SETQ MY-CEN (OUT DRAW-IT 1))
        (SETQ MY-HSZ (OUT DRAW-IT 2))
        (SETQ LOCAL-NEED (OUT DRAW-IT 3)))    ;"Optimal" area need.
;When new son arrives: start him up w/dummy area, place outputs
;on appropriate lists, and make other daemons watch him.
  (ONC (VAL NEW-SON) (DINGS)
    (PROG (SON-NODE NEW-GET)
        (SETQ NEW-GET (OUTPUT (POS (Y ,I-GET) (Y ,I-GET))))
        (SPECIFIES NEW-GET AREA-CHANGE)
        (SETQ SONS-GET (CONS NEW-GET SONS-GET))
        (SETQ SON-NODE (NODE ,NEW-SON NEW-GET MY-CEN MY-HSZ))
        (SETQ SONS-WANT (CONS (OUT SON-NODE 1) SONS-WANT))
        (SETQ SONS-NEED (CONS (OUT SON-NODE 2) SONS-NEED))
        (WATCHES MORE-AREA (OUT SON-NODE 2))
        (OUCH DINGS (CAR SONS-NEED)) ))
;When a request for more area comes from a son: change I-WANT to
;include him, set request flag, and go to father.
  (SETQ MORE-AREA
   (NAMEDONC HIM (VAL DINGS) (I-NEED I-WANT)
        (OUCH I-WANT (MAX LOCAL-NEED (SUM-OF SONS-WANT)))
        (PROG (ISON)
          (SETQ ISON (COND ((EQ DINGS (CAR HIM)) ,DINGS)
                           (T (CAR HIM))))
          (SETQ MY-REQUEST (SON-INDEX ISON SONS-NEED))
          (OUCH I-NEED ,ISON)
          (OUCH DINGA (NOT DINGA)) )))
;When an area change is imposed from father, re-allocate sons
;"normally" if it's not my request (MY-REQUEST = 0),
;otherwise re-allocate giving requesting son all he wants.
  (SETQ AREA-CHANGE
   (ONC (VAL I-GET DINGA) ()
     (COND ((EQ MY-REQUEST 0)
                (NORM-ALLOCATE ,I-GET SONS-GET SONS-WANT))
           (T (SPEC-ALLOCATE MY-REQUEST ,I-GET
                             SONS-GET SONS-WANT SONS-NEED)
              (SETQ MY-REQUEST 0)) ))) )
```

```
(DEFINE NORM-ALLOCATE (IG SG SW)      ;Allocate without special son.
                                      ;IG=I-GET, SG=SONS-GET, SW=SONS-WANT.
  (PROG (TOTW AVAIL LAST) ;TOTal Wants, AVAILable area, LAST Y-edge
    (SETQ TOTW (SUM-OF SW))
    (SETQ AVAIL (- (Y IG) (X IG)))
    (SETQ LAST (Y IG))
    (MAPC (LAMBDA (W G)       ;Wants, Gets -- in proportion to want.
            (PROG (A) (SETQ A (- LAST (* AVAIL (/ TOTQ ,W))))
                      (OUCH G (POS A LAST))
                      (SETQ LAST A) ))
          SW SG) ))


(DEFINE SPEC-ALLOCATE (R IG SG SW SN)
 ;Allocate giving requester his wants.
 ;R=Requester's index, SN=SONS-NEED; other mnemonics as above.
  (PROG (TOTQ AVAIL RN RW LAST)  ;Requester Needs, Requester Wants.
    (SETQ TOTW (SUM-OF SW))
    (SETQ AVAIL (- (Y IG) (X IG)))
    (SETQ RN ,(NTH SN R))
    (SETQ RW (NTH SW R))
    (COND ((< RN (* AVAIL (DIVIDE TOTW ,RW)))
           (NORM-ALLOCATE IG SG SW)
           (RETURN)) )
    (SETQ LAST (Y IG))
    (MAPC (LAMBDA (W G)
            (PROG (A) (SETQ A (- LAST
                                 (+ (* AVAIL (/ TOTW ,W))
                                    (COND ((EQ W RW) RN) (T 0)) )))
                      (OUCH G (POS A LAST))
                      (SETQ LAST A) ))
          SW SG) ))


(DEFINE SON-INDEX (N SN)                  ;Find index to son on list.
        (- (LENGTH SN) (LENGTH (MEMQ N SN)) -1))


(DEFINE SUM-OF (S)                        ;Add up wants or needs of sons.
  (PROG (SUM) (SETQ SUM 0)
              (MAPC (LAMBDA (O) (SETQ SUM (+ SUM ,O))) S)
              (RETURN SUM) ))
```

```
(DEFPIC DRAW-NODE (TXT PA-CEN PA-HSZ I-GET
                   "OUT" MY-CEN MY-HSZ "OUTU" OPTWIDTH
                   "EXTERNAL" DELTAY BEAUTY
                   "AUX" MY-Y PROPOR INWID INHT)
  (SETQ MY-Y (+ (Y ,PA-CEN) DELTAY)) ;Fixed Y coordinate of center.
  (SETQ INWID (/ (TEXTWIDTH TXT) BEAUTY))    ;Width of optimal box.
  (SETQ INHT (/ (TEXTHEIGHT TXT) BEAUTY))    ;Height of optimal box.
  (SETQ PROPOR (/ INHT INWID))          ;Height to width proportion.
  (SETQ OPTWIDTH (/ INWID BEAUTY))              ;Optimal width wanted.
 ;The following gets done whenever I-GET changes.
  (CONTIN
    (OUCH MY-CEN (POS (/ (+ (X ,I-GET) (Y ,I-GET)) 2) MY-Y)
    (PROG (NEWWID USEWID)
      (SETQ NEWWID (- (Y ,I-GET) (X ,I-GET)))
      (COND ((< NEWWID OPTWIDTH)
                        ;If available less than optimum, scale down.
             (SETQ USEWID (* NEWWID BEAUTY))
                                        ;Leave space between boxes.
             (OUCH MY-HSZ (POS (* USEWID PROPOR) USEWID)))
            (T (OUCH MY-HSZ (POS INWID INHT))) ) ) )
 ;Draw the box and the text.
  (TRANSFORM "CENTER" MY-CEN "HALFSIZE" MY-HSZ
        BOXTXT TXT BOXHS)
 ;Draw the fancy line.
  (OUTERLINE MY-CEN MY-HSZ PA-CEN PA-HSZ)  )


(DEFPIC BOXTXT (TXT HALFSIZE)
  (TEXT TXT (POS 0 0))              ;Display text centered at (0,0).
  (PROG (HX HY) (SETQ HX (X ,HALFSIZE))
                (SETQ HY (Y ,HALFSIZE))
                (POLYGON (POS (- HX) (- HY)) (POS (- HX) HY)
                         (POS HX HY) (POS HX (- HY))) )
 ;POLYGON is a function using STATLINE to draw closed polygons.

(DEFPIC OUTERLINE (C1 H1 C2 H2)
                          ;Draw line between outer edges of boxes.
  (LINE (OUT (CLIP C1 C2 C1 H1) 2)    ;See CLIP discussion in text.
        (OUT (CLIP C1 C2 C2 H2) 2)))
```

4.6 Hit-Testing

The machinery so far developed in DALI contains no provision for pointing inputs from graphic input devices, i.e., no way for a user to identify some displayed object by "pointing at it" with a tablet, pen, or other such device. This relates to another deficiency: using DALI as so far defined, it is impossible to make a given object move until it collides with any other object -- not a specific object, but any object at all that happens to be visible. Hit-testing in the manner of Sproull and Sutherland [Spr1, New2] suggests itself as a solution to these problems.

Hit-testing is embedded into DALI in the form of daemons which are run when the visible image produced by any module enters, or leaves, a rectangular area defined by a center position output and a halfsize output. For pointing inputs, the center position can be controlled by the input device, or, alternatively, the tracking dot (cross, arrow, etc.) can be detected by stationary areas.

This aspect of DALI will not be developed in detail. Just the form of DALI-esque solutions for the problems involved will be presented.

On the grounds that they provide a crude tactile sense, daemons watching a rectangular area, defined by two position-valued outputs as mentioned above, will be called touch daemons. Two types of touch daemons are needed, one for detecting objects entering an area and one for detecting exiting objects; the utility of the latter will be seen below. Touch daemons should most generally be a form of named-change daemon, receiving a list of all the objects entering (leaving) their areas. All touch daemons run after all non-touch daemons have run, but the order in which a set of queued touch daemons will run relative to each other is undefined. It is necessary that touch daemons run last in order to make sure they "feel" every entering (leaving) object, especially those in distant, unrelated parts of the picture structure.

The objects received by touch daemons are picture modules, selected by walking up the containment tree from the modules actually "hit" until a module having the designator "NAME" in its picture function's argument list is found. More complex naming schemes could be used, but all must produce picture modules to identify "named objects". Only the descendants of "NAME" modules need be sensitized to hit testing, and functions to sensitize and de-sensitize subtrees should exist.

In some cases, obtaining the "NAME" is unnecessary: stationary function button ("light button") daemons need only know that they were hit, since only the pen module will hit them and their action will usually be to send a simple, constant message to the driving program by, for example, OUCHing an appropriate output. Such daemons need not even be the form of named-change daemon mentioned above.

However, in general a touch daemon wants to know the "NAME" module invading its area, presumably to get some data from him. For example, "light button" testing with a single touch daemon whose area tracks the pen requires that the touch daemon find out what message to send to the driving program. This can be done by getting the appropriate output from the "NAME" module via OUT, getting the output's value with OVAL (,), and sending the value off to the driver as the message.

Continuing interaction with an invading "NAME" module can be achieved by having the touch daemon make a local daemon watch one of the "NAME" module's outputs. The interaction can be cut off when the invader leaves through the action of a touch daemon detecting exiting objects; this is the use of exit detection promised above.

The computation the invading "NAME" module must perform to provide the needed data could be arduous. If the "ships-passing-in-the-night" kind of interaction we are considering will not often happen, the daemons performing the computation could be activated and deactivated. This could be done by daemons whose conditions cause them to run when a daemon becomes, or ceases to be, a "watcher" of a given output. Since the topic of hit-testing is not being considered in full detail, no more

will be said about such daemons; they pose no special problem, but require more scheduling exceptions and elements in outputs.

With the touch daemon operation described here, an interesting situation can occur. Suppose two objects are moving across the screen, each with its own "bumper" touch daemon area surrounding itself, and they simultaneously enter each other's sensitive area. The two touch daemons must be designed to run in either order, of course; but suppose that in both cases the purpose of the "bumpers" is to avoid collisions by veering around blocking objects. Both touch daemons will attempt to "link in" a deviation factor using, for example, center and outer radius outputs from their opposite number's "NAME" daemon -- and the result will be effective data web circularity. However, as will be mentioned in section 5.2, it is a type of effective circularity not requiring "real" data web circularity. Instead, a simple S-DALI mechanism ("future OUCH", see sections 5.2 and 6.2) is appropriate, and can have the result that each object veers only part of the distance either alone would go.

The discussion of hit-testing is essentially concluded; however, it implicitly raises two problems. First, all the simultaneously active touch daemons and "NAME" modules need to be in agreement on the order and meaning of "NAME" module outputs; this is hardly conducive to modularity. Second, typically needed outputs, such as the center and outer radius mentioned above, contain data difficult to gather from sons in a modular fashion.

The first problem could be mitigated by referring to outputs by "name", i.e., by a mutually agreed-upon arbitrary string, rather than by position in the argument list. It was not done due to a possibly misguided desire for efficiency; providing this mechanism would do no great violence to DALI.

The second problem is significantly more difficult to solve. One approach parallels that suggested by Baeker in APPL [Bae1], discussed in

section 1.2: providing user- or system-defined search and composition rules applied recursively down the containment tree from the "NAME" module. For example, if a module doesn't know his outer radius, ask his sons and combine their answers via a daemon specifying a new radius output. System-defined modules, such as LINE, could be defined to "know" the answers to many such standard questions, and could have their type (e.g., "LINE") and inputs made available to allow the user to define new "questions". In some cases, such as the outer radius, answers might be most easily concocted as functions of the actual display buffers. This is a major extension to DALI, and should be approached with some trepiditation.

Chapter 5
Data Web Circularity and Relaxation

5.1 Introduction: The Data Web As a Set of Equations

Circularity in the data web has heretofore been prohibited. This has been accomplished for structurally static data webs by the twin constraints that (1) an output can have only one specifier, and only that specifier can OUCH it; and (2) a daemon may not be created watching an unspecified output. For structurally dynamic data webs, circularity has been prohibited by testing for its presence while adjusting daemon priorities.

Here, data web circularity is discussed, focussing first on when it is really needed, then on what characteristics it should have, and finally on how daemons in a circular data web can be scheduled -- i.e., ordered in their execution. As might be expected, scheduling is far more difficult for cyclic data webs than for acyclic ones.

Before these topics are considered, however, a prior question must be discussed: what is really represented by circularily -- or its lack -- in the data web?

In discussing this question, it is useful to consider the data web as a set of equations. This can be done by: (1) considering each output to be a variable -- particularly including, but not limited to, outputs directly specifying the visible image; and (2) considering each daemon as an equation relating its specified output(s) to some expression(s) written in terms of its watched outputs.

Thus, for example, out simple and familiar RELP picture function

```
(DEFPIC RELP (P1 P2 "OUT" SUM)
    (CONTIN (OUCH SUM (+ ,P1 ,P2))) )
```

since it creates a daemon, adds to the overall set of equations the equation

$$SUM = P1 + P2 .$$

Of course, a complete set of such equations will be difficult to write; for example, there are difficulties in representing daemons which either use and alter the values of local environment variables, or add new equations (daemons) to the set by calling picture functions. These difficulties of representation need not concern us here.

With the data web considered as a set of equations, we can say that what is wanted from a DALI picture definition is a simultaneous solution to all the equations in the set.

The requirement that the data web be acyclic can then be interpreted as meaning that the set of equations are in a form such that they can be solved by back substitution. I.e.: The outputs specified by the driving program are, when DALI begins running, variables whose values are considered known a priori; these outputs may also be considered constants parameterizing the solution desired. The values of these known variables are substituted into equations written only in terms of known variables, a process performed by running the appropriate daemons. This produces more known variables, since after running a daemon its specified outputs can be considered "known". The new known variables are similarly substituted and the process continues until the values of all variables are known. If at any point before the completion of this process we reach a state where no equation is solely in terms of known variables, then two or more equations must cyclicly determine values used by each other -- in other words, the data web contains a cycle.

It is important to note that such cyclic dependence is a function of the particular form of the set of equations, not a function of the represented solution set, i.e., of the desired picture. In many cases

of interest, a cyclic set of equations can be re-written in a form which is not cyclic. As a simple example, the data web representing the equations below is cyclic:

$$X1 = K1 * X2 + C1 \quad (1)$$

$$X2 = K2 * X1 + C2 \quad (2)$$

K1, K2, C2, and C2 are "constants", i.e., outputs specified by the driving process or by some other web ancestor of both of the above-represented daemons.

Without the constraint prohibiting the creation of daemons watching unspecified outputs, these equations correspond to the daemons created by

```
(ONC (VAL X2 C1 K1) (X1)
   (OUCH X1 (+ ,C1 (* ,K1 ,X2))) )

(ONC (VAL X1 C2 K2) (X2)
   (OUCH X2 (+ ,C1 (* ,K1 ,X2))) )
```

This, of course, would create a data web cycle. But by simple symbolic substitution of eq. (2) into eq. (1), we can obtain an alternate form of the equations above which involves no circularity; daemons constructed according to that form produce the same picture, but do not imply a data web cycle.

The task of reformulating the set of equations so that they can be solved by back substitution, if such a reformulation is possible, is the job of the DALI programmer. This should not be considered onerous; it is part of the normal job of a programmer in nearly all situations. What the acyclic daemon scheduling rules of section 3.8 provide is the assurance that for any set of equations which have been correctly reformulated in this manner -- even if elements of this set are added and deleted dynamically -- DALI will produce the correct solution set by substituting (running daemons) in an order which is both correct and efficient.

However, there are situations where it is impossible to reformulate the problem in such a way that back substitution can be performed. Such situations can arise in two ways: (1) The equations to be solved are

simply too complex to be appropriately reformulated; e.g., the equations represent a large set of objects, some fixed and some movable, all simultaneously interacting via an inverse-square law. (2) The equations relating the variables are unknown until run time; e.g., SKETCHPAD.

In such situations, the data web may be cyclic; the precise situations in which it will or will not be cyclic are discussed in the two sections which follow. If the data web is cyclic, then, since the equations involved can be arbitrarily complex and are not represented in a symbolically manipulable form, the only route open to DALI in solving them is to use iterative approximation, or relaxation.

The manner in which cycles are introduced, as well as the daemon scheduling rules involved in relaxation, are the principle topics of this chapter. To anticipate a bit, the general method used for daemon scheduling in a cyclic data web, discussed fully in sections 5.5, 5.6, and 5.7, is this:

First, the cycles are identified, and each separate group of daemons which are all cyclically related to one another is considered a separate composite daemon. When such composite daemons are considered single daemons, the resulting data web is acyclic; so the acyclic daemon scheduling rules of section 3.8 are used for global scheduling, i.e., deciding the order in which entire composite sets of daemons are to be run. The problem of scheduling the daemons within each composite group must then be attacked; unfortunately, it has no uniquely good solution, so several possible alternatives are presented.

However, before cyclic daemon scheduling is discussed in detail, the situations in which it is really needed must be more carefully delineated.

5.2 When Circularity is Not Needed

An important situation in which data web circularity is apparently needed is in the "feedback control" of moving images. For example, consider the problem of moving an object until it touches another object. A natural way to do this is to have a "controlling daemon" C OUCH an output used by a "moving daemon" M to indicate a direction and a rate. M then moves something along, with the current position of the moved object and the current position of the object to be hit watched by a "collision watching" daemon W. When W sees a collision, it wishes to inform the controller, C, so that C can take appropriate action -- e.g., by telling M to stop. Since W is a web ancestor of C, circularity is apparently needed if W is to send its information to C via an OUCH.

However, data web circularity of the type which is the subject of this chapter is neither needed nor particularly appropriate to the situation outlined above. This is the case because when moving images are considered, we are in the realm of S-DALI, not M-DALI, and, as discussed below, the intervention of picture time causes the circularity to assume a different form.

Recall, from section 2.3, that picture time is time as it is intended to be experienced by the viewer; and during DALI compute time -- the time needed to compute each new picture -- picture time is "frozen", i.e., it does not advance.

The preceding section considered the data web as a set of equations whose simultaneous solution is sought; the important point here is that the term "simultaneous" refers specifically to picture time. All the intermediate stages of back substitution, iteration, or what have you are invisible from the point of view of the created picture. The "feedback" situation outlined above generally refers to motion in picture time, i.e., to the relation of successive "frames" of the picture to one another; whereas the processes used to "solve" the data

web serve to define the appearance of each separate "frame" produced, and have no intrinsic effect on the relationship of successive "frames" to one another -- unless, of course, the user has explicitly programmed in such a relationship.

What makes this difference important in the current context is that the mechanisms for relating separate "frames" provide the appropriate means for "feeding back" information in the situation outlined at the start of this section.

As was mentioned in section 2.3, the manner in which separate picture-time frames are related to one another involves the capability of sending messages "across picture time" by means of a "future OUCH", an output value change which the DALI system has been told to perform at some future picture time. The details of "future OUCHes" are covered in sections 6.2 and 6.3.

Now, in the context of S-DALI, separate picture times, and future OUCHes, variables in the equation model of the data web must be subscripted to indicate which picture time they refer to. Thus, the pair of circular linear equations in section 5.1, eqs. (1) and (2), become

$$X1(t) = K1 * X2(t) + C1 \quad (3)$$

$$X2(t) = K1 * X1(t) + C2 \quad (4)$$

Here, K1, K2, C1 and C2 have not been subscripted because they can be considered constants. The above equations are circular because the time subscript of all uses of X1 and X2 is the same. On the other hand, if we wish to represent daemons "future OUCHing" outputs for some future picture time, the equations become:

$$X1(t+i) = K1 * X2(t) + C1 \quad (5)$$

$$X2(t+i) = k1 * X1(t) + C1 \quad (6)$$

There is no circularity involved in the above equations because each equation refers to an element of the picture-time sequence different from the one which its partner defines.

In this way, future OUCHes can be used to create effective data web

cycles containing a non-zero picture time delay. This delay can be very small -- down to the resolution limit in the picture time domain -- but it must exist.

By virtue of the picture time delay involved, future OUCHes are the appropriate means for providing "feedback control" as referred to at the start of this section. Referring to that example, the "controller" C directly OUCHes outputs watched by the "mover" M, and M directly OUCHes outputs watched by the "collision watcher" W. However, W can "future OUCH" an output watched by C to provide the desired feedback, causing this OUCH to occur at a picture time before or equal to the next incremental move of M. When the "future OUCH" occurs, C will run before M since it is M's web ancestor; thus if C directly OUCHes a "halt" order to M, motion will stop instantaneously.

Relaxation can be performed in a similar manner. In this case in particular, the non-zero delay restriction has the effect of causing intermediate, non-stable states of the relaxation to be visible: the viewer watches relaxation in motion. This occurs because at each iteration, the entire operation of M-DALI is performed. Hence, for example, intermediate OUCHes of the inputs to LINE modules will cause them to change the visible display. This is not always undesirable; it is, for example, the mode of operation used by SKETCHPAD.

It is true that the inherent delays involved in this type of feedback can slow convergence and cause instabilities -- divergence and/or oscillation. However, such instabilities are characteristics of the system being simulated rather than characteristics of DALI, since the picture time delays involved are fully under the control of the user.

5.3 When Circularity Is Needed

When, then, is true data web circularity really needed? It is needed under three conditions:

(1) The output values defining a picture are most readily computed by some iterative method.

(2) It is inconvenient or impossible to perform this iteration within the body of a single daemon.

(3) The iteration is performed only for its effect, and thus its internal dynamics are not of interest. I.e., it is not desirable to have the picture reflect the intermediate states of the iteration and show the relaxation in motion.

The first condition is was discussed in section 5.1. As mentioned there, this situation occurs when the system defined cannot be reformulated to allow back substitution, either because it is too complex or because it is insufficiently defined until run time.

The second condition alludes to the fact that it is entirely possible to perform iterative operations by looping within a single daemon body. In some sense, this is "cheating", since it uses the Turing ability of the base language to avoid using DALI proper; nevertheless if this is most convenient, there is no reason to shy away from it. However, it may be decidedly inconvenient for two reasons:

In the first place, it is impossible to use previously defined picture modules and daemons -- as opposed to previously defined functions -- to perform part of a loop within a single daemon. This follows from two facts: daemon executions are not nested within one another, and a daemon cannot exert direct control over the DALI compute time at which another daemon runs.

Second, the iterating daemon must be explicitly written to provide

for all the possible cyclic dependence which can exist among its data. This can be particularly difficult when such dependence is to be dynamically created, e.g., as in SKETCHPAD.

The third condition required for web circularity -- the fact that intermediate states are "uninteresting" and should not be visible -- could be achieved with future OUCHes and mechanisms for delaying visible changes until the iteration terminates. However, this implies a loss of efficiency over what could be attained since many more daemons will generally be run per iteration cycle than are actually needed. Consider, for example, Fig. 5-1, a graph showing ancestry relations in a hypothetical data web containing a loop four daemons long; the square box marked L is the special loop daemon, to be described later, which effects the loop. If this loop is effected by future OUCHes, all 16 daemons in the figure will be run on each iteration. If intermediate states need to be visible, this is necessary; but if intermediate states do not need to be visible, only the four daemons actually taking part in the iteration need be run in each cycle.

The possibility of much greater efficiency is the interesting issue with regard to cyclic data webs. It should not, therefore, come as a great shock that the methods to be introduced for dealing with cyclic data webs resemble certain code optimization techniques used in language compilation.

5.4 Introducing Cycles: Loop Daemons

To introduce cycles in the data web, the programmer must make use of a new type of daemon: the loop daemon. Loop daemons are created and returned as the value of

(LOOPON cndtn (-specs-) -body-)

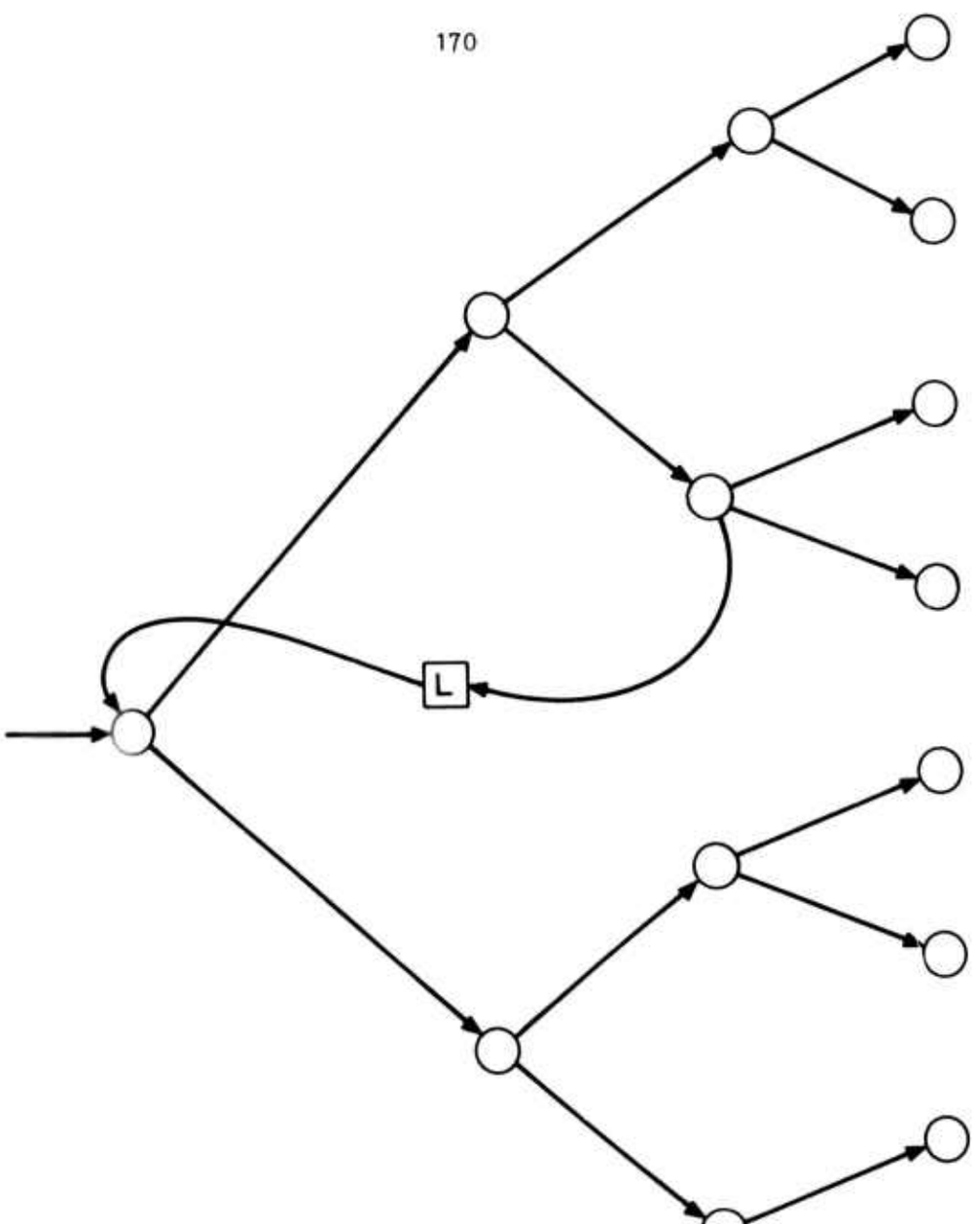


FIGURE 5-1 INEFFICIENCY OF "FUTURE OUCH" FEEDBACK

where cndtn is the condition, (-specs-) is a list of specified outputs, and -body- is the body of code to be run. A "namedloopon" daemon could also exist, but will not be described. Loop daemons with conditions which do not watch outputs, e.g., DELETE, do not make any sense; so they are illegal.

Loop daemons differ from normal daemons in that when they are created, the outputs they watch need not be specified. They may thus be used to create data web cycles. Creation of cycles by use of WATCHES and DEPENDS is also legal, providing every cycle contains at least one loop daemon. This condition will be checked by DALI; the check can be performed by an analog of RE-PRIORITIZE, although a more efficient method will be presented in section 5.8.

With loop daemons, the connectivity of the data web is effectively arbitrary. In particular, allowing more than one daemon to "specify" an output is unnecessary. The MERGE module presented in section 4.4, which OUCHes its output to the value of the most recently OUCHed of its two inputs, can be used to create the same effect. Having to create such MERGE modules can, however, be highly inconvenient; the ability to multiply specify an output should be supplied by a scheme which is equivalent to automatic creation of MERGE outputs by DALI. While this is straightforward, multiple specifiers will not be assumed since they would unnecessarily complicate the upcoming discussion of scheduling.

The rationale for introducing a special type of daemon for the creation of data web cycles, rather than just allowing the creation of cycles with WATCHES and DEPENDS, is twofold:

First, if loops are created, some mechanism for terminating the implied iteration must be present. No explicit mechanism for this is provided by DALI. Instead, since execution is propagated by means of OUCHes, one daemon in each loop must perform a test and simply not do a critical OUCH if adequate convergence has been achieved. Loop daemons provide a convenient and obvious place, although not a necessary one, to

perform convergence testing. Furthermore, since DALI requires that at least one loop daemon be in every loop, the user is guaranteed to have enough tests to terminate every loop. In any event, loop daemons serve to remind the user that such tests are necessary.

The second reason is more interesting. In an acyclic data web, a non-loop daemon can be viewed as establishing an invariant relationship between its watched outputs and its specified outputs; the acyclic scheduling rules were chosen to allow this important interpretation to be made, and a scheduling rule will be chosen to keep this interpretation valid for non-loop daemons in cyclic data webs. However, this interpretation cannot be maintained for all daemons in generally cyclic data webs; if it were maintainable, everything would be correct on the first pass through and no iteration would be necessary! So, since they are already somewhat special, loop daemons are chosen as not strictly maintaining relationships between watched and specified outputs. Characterizing what they actually do is rather more difficult -- they certainly are not approximations to static relationships -- and will not be attempted.

5.5 Goals of the Cyclic Daemon Scheduling Rules

The daemon scheduling rules for a cyclic data web, which will be presented in section 5.7, are motivated by two goals: acyclic compatibility and efficiency.

Acyclic compatibility means that an acyclic subweb is guaranteed to "do the same thing" whether or not it is embedded in a cycle. This can be stated more precisely as follows: Let W be a subweb of a cyclic data web C such that W contains no loop daemons. Then, under any pattern of initial OUCHes, W embedded in C must exhibit that behavior guaranteed to W if it were removed from C and run under the previously specified

acyclic daemon scheduling rules. The term "subweb" is precisely defined in the next section.

Efficiency is given the following interpretation: if a daemon D watches an output O, and D is not part of any data web cycle including the specifier of O, then: D must not run until there is no possibility of O's being OUCHed again. Thus, if a daemon uses an output's value and does not participate in its specification, it sees only the final, converged value of the output. Thus, no relaxation of an output's value can affect how efficiently that value is used after it has been fully defined.

"Efficiency" is perhaps the wrong word to use in denoting what is described above, since it has a further, important effect: Entire cyclic subwebs, as opposed to individual daemons within cyclic subwebs, can be viewed as establishing invariant relations between output values "entering" the subweb and output values "leaving" the subweb. Furthermore, the process of establishing these relations is invisible to a daemon which just uses them.

5.6 Preliminary Definitions

Several definitions are needed in the statement of the cyclic daemon scheduling rules. The first two were presented in section 3.8, and are repeated here for reference:

> A daemon A is a web father of a daemon B, and B is a web son of A, if and only if A specifies an output watched by B.
>
> A daemon A is a web ancestor of a daemon B, and B is a web descendant of A, if and only if there exists a sequence of daemons D(0),D(1)...,D(n) such that D(0)=A, D(n)=B, and for $0<i<n+1$, D(i-1) is a web son of D(i).

In addition, the concepts of acyclic daemon, subweb, and maximal strongly connected subweb (MCS) are needed. They are defined as follows:

> An acyclic daemon is a daemon which is not its own web ancestor.
>
> A subweb S of a data web W is a set of daemons D in W, and a set of web father and web son relations R such that R contains all the web father - web son relations in W between elements of D, and only those relations.
>
> A maximal strongly connected subweb (MCS) is a largest subweb containing only daemons who are all web ancestors of each other.

Daemons in intersecting data web cycles, i.e., cycles with a daemon in common, are members of the same MCS; and every daemon is either an acyclic daemon or a member of some MCS.

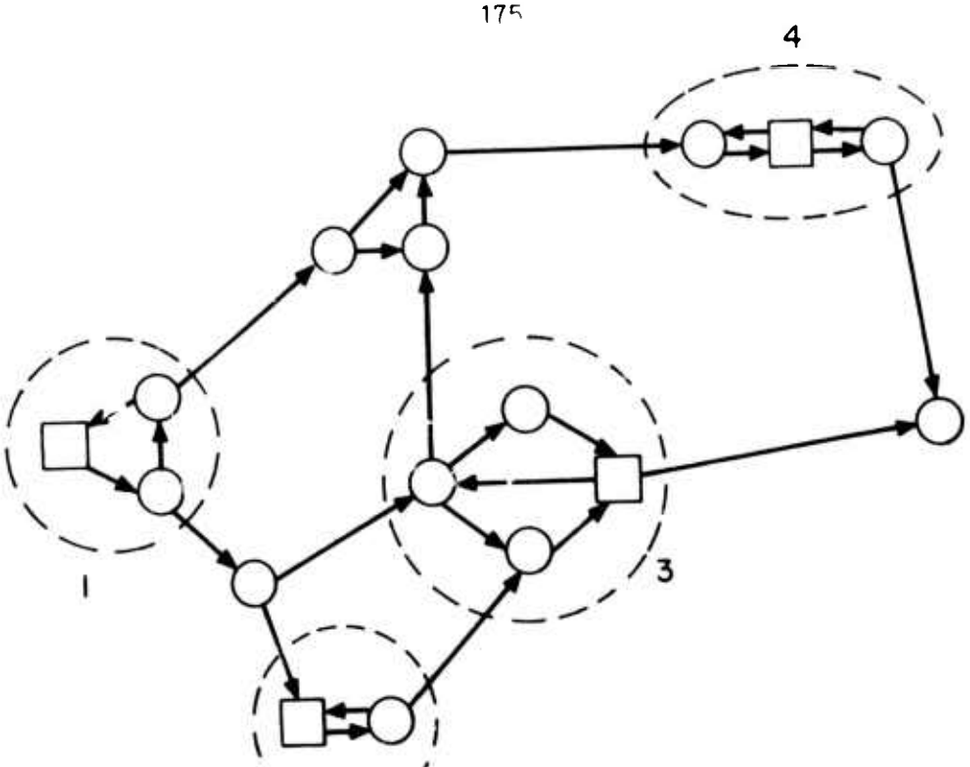
The concept of an MCS is crucial to cyclic scheduling, in that all the elements of an MCS must participate in the "relaxation" of that MCS, and no daemons outside an MCS need participate in its "relaxation".

It is of particular note that all the daemons in an MCS have identical web ancestral qualities: a web ancestor (descendant) of any daemon in an MCS is a web ancestor (descendant) of every daemon in that MCS. This is true by the definition of an MCS above; it is also true of any isolated cycle in the data web.

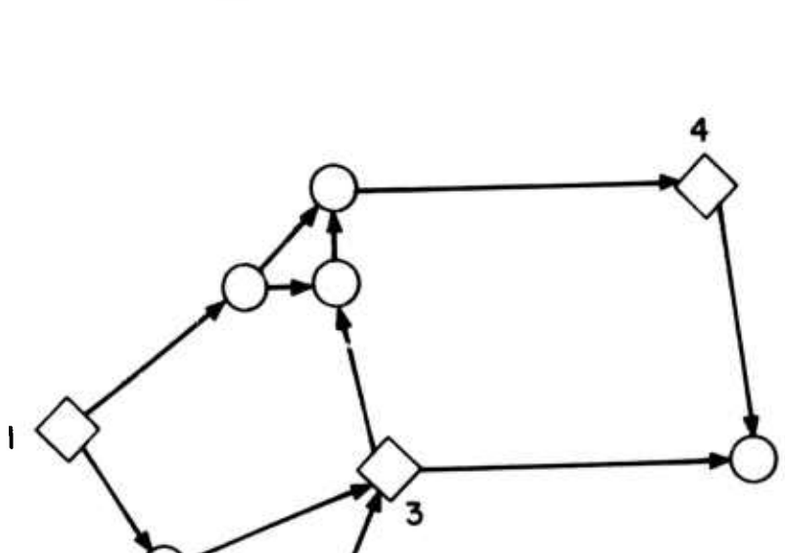
As a result of the web ancestral qualities of an MCS, it is convenient to speak of the web ancestors and descendants of an entire MCS, treating it as if it were a single composite daemon.

The effective data web formed by merging MCSs into single composite daemons is acyclic. If it were not, then two daemons or composite daemons would exist which are mutual ancestors and not in the same MCS; this contradicts the fact that each MCS is "largest".

The creation of an acyclic data web by making MCSs into composite



A



B

FIGURE 5-2 ACYCLIC DATA WEB VIA COMPOSITE DAEMONS

daemons is illustrated in Fig. 5-2, which, like Fig. 5-1, shows only father-son ancestry relations represents loop daemons as squares. Fig. 5-2a shows the original, hypothetical data web containing four MCSs; Fig. 5-2b shows the acyclic data web resulting from collapsing the MCSs, with each composite MCS daemon shown as a diamond.

A final definition:

> A daemon A is an acyclic web ancestor of a daemon B if and only if there exists a sequence of daemons D(0),D(1),...D(n) such that D(0)=A, D(n)=B, for every i in the range $0<i<n+1$ D(i-1) is a web son of D(i), and for every i in the range $0\leq i\leq n$ D(i) is not a loop daemon.

Note that no loop daemon can be an acyclic web ancestor of any daemon. The above definition of an acyclic web ancestor will be used in assuring acyclic compatibility.

5.7 The Cyclic Daemon Scheduling Rules

The daemon scheduling rules for the general case of cyclic data webs, called the cyclic daemon scheduling rules, are divided into three groups: (1) rules governing the local activities of selection and noninterruption; (2) rules achieving the previously stated goals of efficiency and acyclic compatibility; and (3) rules governing the behavior of loop daemons.

Whenever the term "daemon" is used below without qualification, it refers to both loop and non-loop daemons.

The first group of rules is taken unchanged from the acyclic case of section 3.8:

Rule 1: (selection) A daemon will be run if and only if one or more of its watched outputs has been OUCHed; once run, it does not run again until such an OUCH occurs again.

Rule 2: (noninterruption) Once a daemon D begins execution, no daemon web-ancestrally related to D may run until D terminates of its own accord.

The next group of rules consists of two rules achieving the goals of efficiency and acyclic compatibility.

Rule 3: (efficiency) If daemons A and B are to be run, A is run before B if A is a web ancestor of B and B is not a web ancestor of A.

Rule 4: (acyclic compatibility) If daemons A and B are to be run, and A is an acyclic web ancestor of B, then A runs before B.

By virtue of the web ancestral qualities of maximal strongly connected subwebs (MCSs), Rule 3 establishes an "ancestors before descendants" rule in the acyclic data web formed by considering MCSs as composite daemons. In this way, Rule 3 guarantees efficiency in the sense discussed previously.

Rule 4 straightforwardly establishes acyclic compatibility. It cannot conflict with Rule 3, since if a daemon A is an acyclic web ancestor of a daemon B, then A must also be a web ancestor of B.

The first group, Rules 1 and 2, cover aspects of scheduling which are local, affecting the operation of individual daemons independent of their relationships with other daemons. Rule 3, efficiency, covers global aspects of scheduling, ordering the operation of maximal strongly connected subwebs (MCSs) and acyclic daemons with respect to one another. The problem these rules leave is that of ordering the operation of daemons within an MCS with respect to one another.

The rules governing the operation of daemons in an MCS are most easily comprehended if MCSs are redrawn in the manner shown in Fig. 5-3,

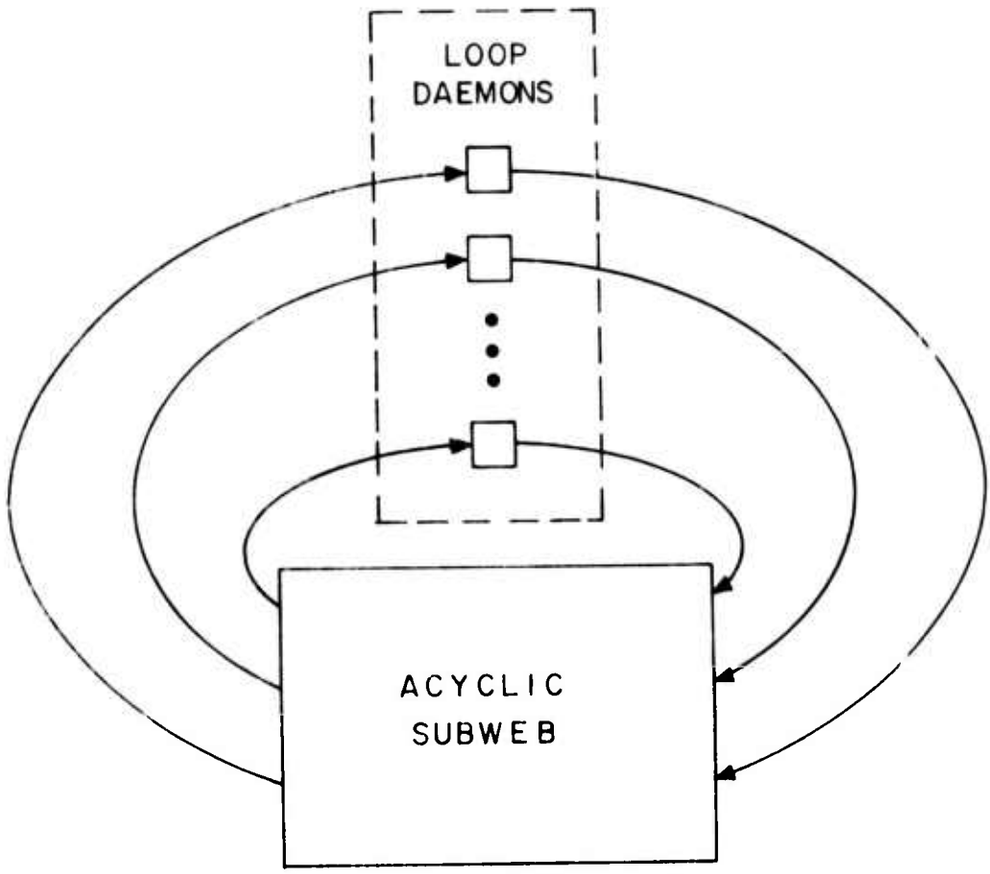


FIGURE 5-3 GENERAL FORM OF AN MCS

separating loop daemon "feedback paths" from the rest of the daemons in the MCS. Since every cycle must contain a loop daemon, the subweb remaining when loop daemons are separated out must be acyclic. As in the previous figures of this chapter, Fig. 5-3 shows only father/son ancestry and not the intimate details of the data web. Each of the ancestry arrows of Fig. 5-3 can be multiple, as illustrated for a hypothetical data web in Fig. 5-4.

The daemons in the acyclic data subweb of these figures are the province of Rule 4, acyclic compatibility; that rule orders the operation of these daemons with respect to each other. Thus, we are ultimately left with the problem of scheduling loop daemons in a given MCS with respect to one another and with respect to the non-loop daemons in the MCS.

Three different methods of scheduling loop daemons will now be presented, each of which has advantages and disadvantages. The three methods are derived from the Gauss-Seidel and the Jacobi iterative techniques for solving systems of equations [Ral1, Var1]; the applicability of these techniques follows from considering the data web as a set of equations, as discussed in section 5.1.

The first and third methods are analogous to Gauss-Seidel iteration, sometimes called the method of successive displacements, in that only one loop daemon "feedback path" is exercised on each iteration. The first is easy to implement, but inefficient in that it produces situations where daemons are run without use being made of the results they produce. This inefficiency does not occur in the third method; but the third method is difficult to implement.

The second method is analogous to Jacobi iteration, in that many loop daemon "feedback paths" can be exercised on each iteration. Jacobi iteration is also variously called the method of simultaneous displacements, the point total-step method, and the point Jacobi method. While relatively easy to implement and not inefficient in the manner of

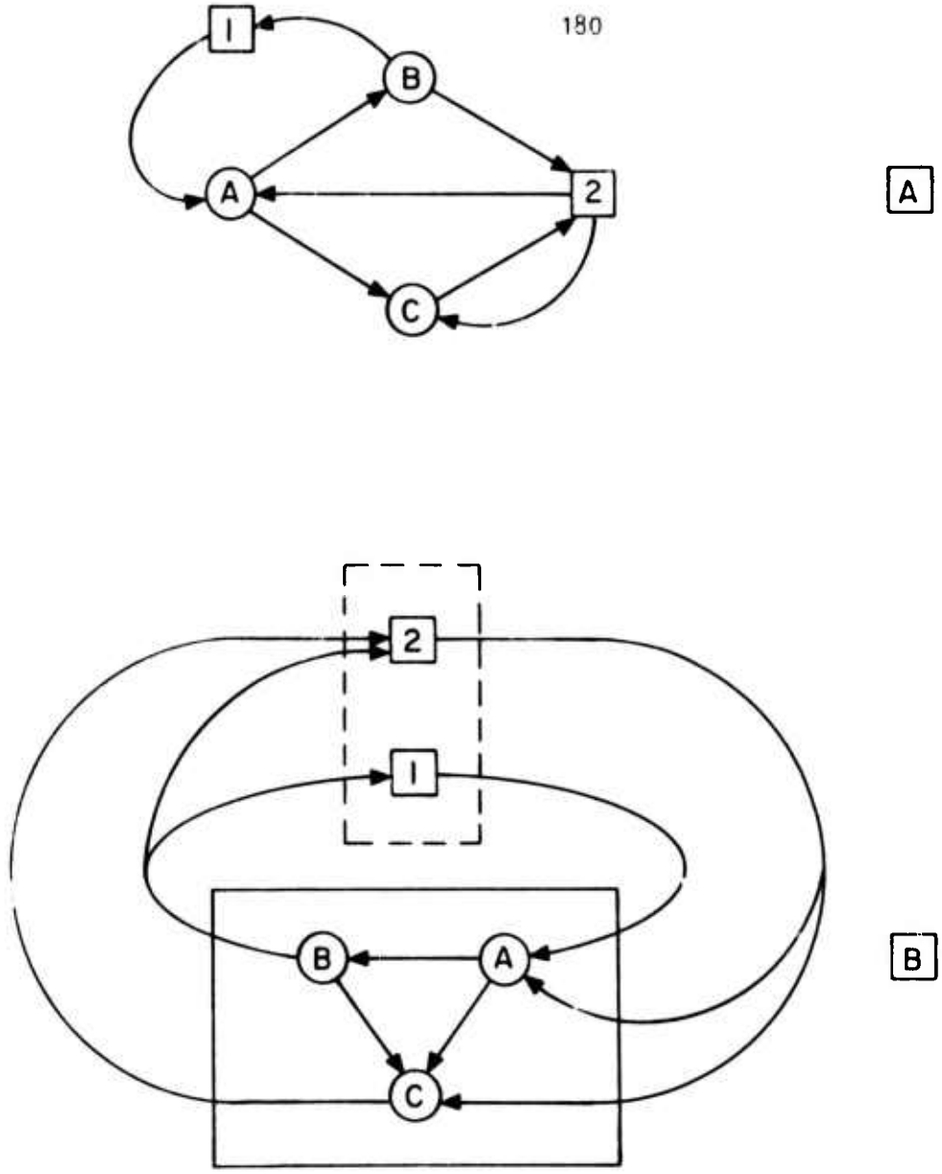


FIGURE 5-4 REDRAWING IN GENERAL FORM

the first Gauss-Seidel method, Jacobi iteration *per se* often converges more slowly than Gauss-Seidel, and often does not converge when Gauss-Seidel does [Ral1, Var1]. This was noted by I. Sutherland in [Sut1], and led to the use of Gauss-Seidel iteration, although not by that name, in SKETCHPAD.

However, the question of convergence is complex. There are systems of linear equations for which Jacobi iteration converges and Gauss-Seidel does not, especially when the linear equation system coefficient matrix is not positive; an example can be found in Chapter 3 of [Var1]. SKETCHPAD, in its use of linear approximations to constraints, is much closer to the classical case of iterative solutions to systems of approximating linear equations for which Gauss-Seidel iteration is generally superior; hence Gauss-Seidel iteration was the clear choice. The equation system effectively implemented by a DALI data web, on the other hand, need not be linear or even constant in time. DALI programs can clearly be written which will converge only if one or the other method is used. Hence, a choice of a set of scheduling rules that is good for all cases does not appear possible. Furthermore, the rules given for "Jacobi" iteration actually produce Gauss-Seidel iteration in some circumstances, as will be pointed out.

Inefficient Gauss-Seidel scheduling (IGS) is performed by these rules:

Rule 5I: (*non-loop before loop*) If a non-loop daemon N and a loop daemon L are to be run, and they are mutual web ancestors, then N runs before L.

Rule 6I: (*loops alternate*) If loop daemons A and B are to be run, the least recently run of A and B is run before the other.

Rule 7I: (*closure*) In any cases not otherwise covered, daemons may be run in any order.

For the purposes of Rule 6I, loops alternate, loop daemons which have never been run are considered to have been run at random and

distant past times. This rule simply assures that all the loop daemons in an MCS which are involved in the iteration by virtue of Rule 1, selection, will affect the iteration in a fairly equitable fashion. Since there must be a loop daemon in each loop, this means that all loops in an MCS will share in the iteration.

The effect of Rule 5I, non-loop before loop, is to cause only one loop daemon to be run for each iteration. This occurs because executing a loop daemon under Rule 6I will generally cause some non-loop daemons in the acyclic subweb of an MCS to wish to run; and by Rule 5I, these will take precedence over any other loop daemons.

The inefficiency of this scheduling method arises from the fact that there is no guarantee that the one loop daemon who does run on each iteration will use, or can use, all the results which the MCS's acyclic subweb has generated. As an example, the daemons of the MCS in Fig. 5-4 will typically run as follows under IGS:

A,B,C,1,A,B,C,2,A,B,C,1, . . .

Every other execution of daemon C, namely those runs followed by running loop daemon 1, is redundant in the sense that whatever values C computed are not used before running C again.

Jacobi scheduling (JS) is performed by this set of scheduling rules:

Rule 5J: (feedforward) As long as a non-loop daemon can be run without violating Rule 3, efficiency, no loop daemon is run.

Rule 6J: (feedback) When no non-loop daemon can be run without violating Rule 3, all of the loop daemons which (a) currently wish to be run, and (b) can be run in any order without violating Rule 3, are all run once in an undefined order before running any other daemon.

Rule 7J: (closure) In any cases not otherwise covered, daemons may be run in any order.

The effect of Rules 5J, feedforward, and 6J, feedback, is to

alternate between (a) running all daemons in the acyclic subweb of an MCS, and (b) running all loop daemons which are to be run as a result of (a) or the previous (b). Thus under JS, the MCS of Fig. 5-4 might will typically run as follows:

A,B,C,1,2,A,B,C,1,2, . . .

Clearly, JS does not have the inefficiency ascribed to IGS. However, examples where JS does not converge and IGS does are easy to find; one is presented below.

Consider for example the simple data web of Fig. 5-5. Suppose every daemon in the figure simply copies its watched output into its specified output; initially all daemons are queued and the initial output values are A=1, B=2, C=3, and D=4. Then, JS would run as shown in the following table. In that table, time proceeds vertically downwards, values of outputs are shown in the right-hand columns, and the daemons JS runs are shown on the left; each daemon utilizes the output values shown on the preceding "value line" and produces the output values on the succeeding "value line".

daemons run	A 1	B 2	C 3	D 4
N1 and N2	4	2	2	4
L1 and L2	4	4	2	2
N1 and N2	2	4	4	2
L1 and L2	2	2	4	4
N1 and N2	4	2	2	4
(loop to second state)				

The last line is the same as the second, so we have an infinite loop -- oscillation. Convergence testing anywhere will not help, since the initial values can be made different enough to overcome any fixed bound.

Under the same initial conditions, IGS converges. A possible trace, using the conventions of the previous table, is:

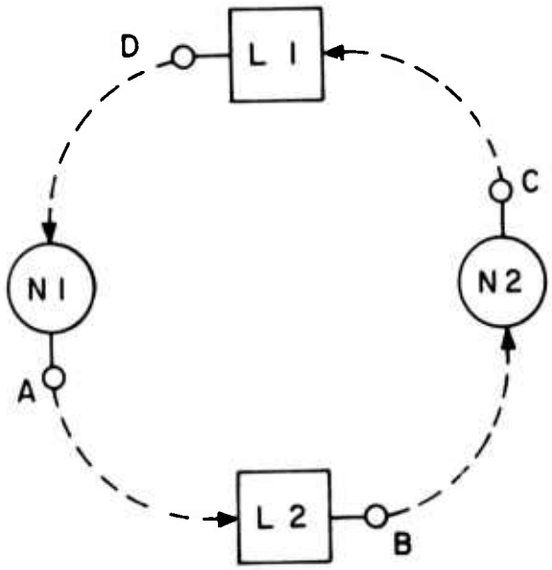


FIGURE 5-5 IGS CONVERGES,
JS OSCILLATES

daemons run	A 1	B 2	C 3	D 4
N1 and N2	4	2	2	4
L1	4	2	2	2
N1	2	2	2	2
L2 (terminates)				

Any reasonable convergence test will halt the process when L2 runs. The arbitrary choice of L1 as the first loop daemon run causes the equally arbitrary selection of 2 as the final value, rather than 4. Either value is a possible solution to the system defined under the given initial conditions. If one of the daemons halves the value it receives, both schemes converge to A=B=C=D=0; interestingly, in this latter case JS requires approximately twice as many daemon executions as IGS.

Use of the term "Jacobi scheduling" to describe the above scheduler is not precisely correct, since each loop daemon has access to most recently computed values of outputs. This is especially true when one loop daemon is a web son of another in the same MCS, with no intervening non-loop daemons. If both the "father" and "son" are to be run in the same feedback session, and the father happens to run first, the son will indeed see most recently computer values and the effective iteration will correspond most closely to Gauss-Seidel iteration. However, such ordering of daemons is completely providential, and the existence of intervening non-loop daemons always produces the effects of Jacobi iteration.

Now an "efficient" Gauss-Seidel scheduler (EGS) will be defined which only runs one loop daemon per iteration but runs no non-loop daemon unnecessarily. Intuitively, we wish to run only those non-loop daemons in the acyclic subweb of an MCS whose operation can effect the loop daemon which will next be run; those non-loop daemons are just those daemons which are "ancestors" of the <u>target</u> loop daemon, in a sense of ancestry which does not propagate through loop daemons.

To formalize these notions, some further definitions are needed.

A daemon A is a direct ancestor of a daemon B, and B is a direct descendant of A, if and only if there exists a sequence of daemons D(0),D(1),...D(n) such that: D(0)=A and D(n)=B; for each i 0<i<n D(i) is a web father of D(i+1); and for 0<i<n no D(i) is a loop daemon.

Note that in the definition of direct ancestor and descendant, both A and B may be loop daemons; this is the difference between a direct descendant and an acyclic descendant.

The specification of a "target" loop daemon is in terms of a set of "target" loop daemons, defined as follows:

The target set T is a set of loop daemons such that:

(a) All members of T are web ancestors of every member of T, i.e., all members of T are in the same MCS.

(b) The members of T are all direct descendants of some daemon(s) which are to be run.

(c) There is no other possible target set T' such that members of T' are web ancestors of members of T.

By virtue of condition (b), T is dependent on the daemons which are currently "queued" to be run. In a direct implementation, T would have to be modified whenever a daemon was run or queued.

T can be empty. This can occur trivially in an acyclic data web, since it contains no loop daemons; it will also occur in a cyclic data web when all MCSs have finished their iteration.

Whenever T is nonempty, there is a target daemon t:

The target daemon t is some member of T which has least recently been the target daemon.

A daemon which has never actually been the target daemon is considered to have been the target daemon at some randomly chosen time the remote past.

The rules for "efficient" Gauss-Seidel scheduling (EGS) are:

Rule 5E: (aim for target) If there is a target daemon t, the only

daemons which may run are t and non-loop daemons which are direct ancestors of t.

Rule 6E: (target last) If daemons A and B are to be run, and both can be run by Rule 5E, and A is a direct ancestor of B, then A runs before B.

Rule 7E: (switch targets) If it exists, the current target daemon ceases to be the target daemon as soon as either (a) it is run, or (b) as a result of Rule 5E, no daemons can be run.

Rule 8E: (closure) In any case not otherwise covered, daemons may run in any order.

Rule 5E, aim for target, produces the "efficiency" of EGS. It is also the reason why "efficiency" has been put in quotation marks; the author can conceive of no way of implementing it in an even marginally efficient fashion.

Rule 7E, target last, clearly cannot conflict with Rule 3, efficiency, or Rule 4, acyclic compatibility. As its name suggests, Rule 6E exists only to guarantee that the target daemon is run after all the non-loop daemons affecting its watched outputs are run.

Clause (b) of Rule 7E, switch targets, is needed to allow for the fact that no daemon need OUCH all its outputs. This can lead to a situation where all paths to the target daemon are "blocked" and the system is stymied.

5.8 Implementation of the Cyclic Daemon Scheduling Rules

The implementation discussed below is couched in terms of implementing the "Inefficient Gauss-Seidel" scheduler (IGS) presented in the previous section. Implementation of the "Jacobi" scheduler (JS) is very similar; it is discussed as a modification of the IGS implementation. Implementation of the "efficient" Gauss-Seidel

scheduler (EGS) will not be discussed, as no inefficient implementation of EGS has been formulated.

Implementation of IGS and JS can be done with a priority scheme similar to that used for the acyclic scheduling rules. We again have a daemon queue containing all daemons yet to be run, with daemons queued in the order in which they are to be run. Rule 1, selection, and Rule 2, noninterruption, are satisfied by queueing a daemon when any of its watched outputs are OUCHed. The order of daemons in the queue is determined from two priority numbers associated with each daemon: its cyclic priority and its acyclic priority. The determination and use of these priorities, and the manner in which the daemon queue is used, constitutes the implementation.

The acyclic priority of a daemon is defined as follows:

(1) The acyclic priority of the driving process is 0.

(2) The acyclic priority of a loop daemon is initially -1, and, for IGS, is adjusted downward during execution; in JS, it remains -1 forever.

(3) The acyclic priority of a normal daemon is 1 if it has no normal daemon web fathers; otherwise it is at least 1 greater than the largest acyclic priority of its web fathers.

The acyclic priority of a daemon is easily initially determined at the daemon's creation. Structural changes to the data web, caused by SPECIFIES, WATCHES, and the like, (section 4.4) may result in a need to adjust non-negative acyclic priorities. This can be done with a RE-PRIORITIZE modified to return when it encounters a loop daemon; in this way, the test for circularity now assures that at least one loop daemon resides in each loop. Another method of re-specifying acyclic priorities and assuring that loops contain loop daemons will be discussed later.

In IGS, the acyclic priority of a loop daemon is readjusted to be

the same as the value of a global variable LOOPLAST each time the loop daemon is run. LOOPLAST is initially -2 and is decremented by 1 every time it is used. Positive acyclic priority daemons are queued before negative acyclic priority daemons, thereby satisfying Rule 5E, non-loop first. By queueing daemons whose acyclic priorities are the same sign in inverse order of the magnitude of their acyclic priorities, Rule 6E, loop alternate, and Rule 4, acyclic compatibility, are observed.

In determining cyclic priorities, we wish to establish the following situation:

(1) The cyclic priority of the driving program is 0.

(2) The cyclic priority of all daemons in a given maximal strongly connected subweb (MCS) is the same.

(3) The cyclic priority of any daemon D is greater than that of any of its web ancestors not sharing an MCS with D.

This causes the cyclic priorities to be analogs of the priorities used in the acyclic case, established in the acyclic data web where each MCS is considered a single composite daemon. By having cyclic priorities override acyclic priorities in the queue ordering, Rule 3, efficiency, will be observed.

Unfortunately, the determination of cyclic priorities as described above cannot be done on a purely local basis, as could the determination of acyclic priorities. Instead, we must start from the connectivity matrix C of the data web: C is a boolean matrix, and C(i,j)=1 if and only if daemon i is a web father of daemon j. We assume some indexing scheme to associate each daemon with its own row and column of C. The construction of C can be performed in time linear with the number of daemons by the recursive algorithm MAKE-C below.

MAKE-C is applied to a daemon D:

For each daemon d dependent on an output specified by D, do the following:

(1) If row d of C contains any non-zero entries, set a flag F false; otherwise set F true.

(2) Set C(D,d) to 1.

(3) If F is true, apply MAKE-C to d.

Applying MAKE-C to the driving program will construct C. Alternatively, C can be incrementally created and modified as daemons are created and destroyed, SPECIFIES is used, etc. An incrementally extensible C will require further spatial overhead, however.

Having C, the MCSs of the data web can be determined.

First the boolean reachability matrix R of C is constructed: R(i,j)=1 if and only if daemon i is a web ancestor of daemon j. R can be found as the limit of (C+I)**N as N increases, where I is the identity matrix and ** indicates exponentiation. Ramamoorthy [Ram1] gives an alternative algorithm for constructing R which is faster than direct matrix multiplication; it will not be discussed.

Now the symmetric matrix M is constructed, after [Ram1]. M is the elementwise intersection (logical AND) of R and R transpose, the latter being the reaching matrix of C. M may as well stand for Magic: every row of all zeros corresponds to a daemon not in any loop; every distinct non-zero row corresponds to a different MCS; and each non-zero row has a 1 in each column corresponding to a daemon in that MCS [Ram1].

Given M, we can mark each daemon with an integer which either uniquely identifies its MCS or indicates that it is not in any MCS, i.e., it is an acyclic daemon. Then a procedure similar to RE-PRIORITIZE can assign cyclic priorities. As was mentioned in section 4.4, however, the time to RE-PRIORITIZE can rise exponentially with the number of daemons. Given C, we can do better than that; the algorithm below runs in time linear with the number of MCSs and acyclic daemons.

First, construct ACT, the transpose of acyclic connectivity matrix which considers each MCS to be a single composite daemon. ACT has a row and column for each acyclic daemon and each MCS, and ACT(j,i) is:

if i represents an acyclic daemon p:

if j represents an acyclic daemon q, ACT(j,i) is C(p,q).

if j represents an MCS m, ACT(j,i) is the OR of all C(p,q) for each q in m.

if i represents an MCS m:

if j represents an acyclic daemon q, ACT(j,i) is the OR of C(p,q) for each p in m.

if j represents an MCS n, ACT(j,i) is the OR of C(p,q) for each p in m and q in n.

ACT can be easily constructed by first constructing a mapping vector from an index of ACT to an index of C (for acyclic daemons) or to a list of indices of C (for MCSs). This same mapping vector will be useful in the actual priority assignment below.

Now priorities are assigned, using an integer P and a boolean vector X. X has as many elements as each row and column of ACT. P is initially 0, and X is initially all 0's. Priority assignment is performed by the following three-step sequence, repeated until the test in step (1) signals completion:

(1) Find a row i of ACT which is all zero such that X(i)=0. If no such row exists, return.

(2) If i corresponds to an acyclic daemon, give it cyclic priority P; if it corresponds to an MCS, give all its members cyclic priority P.

(3) Set column i of ACT to all 0's, set X(i) to 1, and increase P by 1.

This must assign a cyclic priority to a daemon or MCS that is greater than the cyclic priorities of its ancestors, since P is ever-increasing and a cyclic priority is not assigned to a daemon or MCS until after all its ancestors have been assigned cyclic priorities. It must initially find a valid i, since the effective data web it operates or is acyclic and some node in an acyclic graph must have no ancestors. Since removing a daemon from an acyclic data web leaves an acyclic data web, the algorithm cannot terminate until it has assigned a cyclic priority to every daemon.

An identical priority assignment algorithm could be used to assign acyclic priorities by using, instead of ACT, the transpose of C with the

rows and columns representing loop daemons either deleted or ignored. With this method, the requirement that every loop contain a loop daemon can be checked by examining the vector X after the algorithm terminates; if X contains any zeros, the daemons corresponding to the zero entries constitute one or more loops with no loop daemon.

The rules for ordering daemons in the daemon queue are:

Given daemons i and j, with cyclic priorities CPi and CPj and acyclic priorities APi and APj, then

(1) If CPi < CPj, i is queued before j.

(2) If CPi > CPj, i is queued after j.

(3) If CPi = CPj, then

If APi = APj, i and j may be queued in either order.

If APi and APj are both positive or both negative, the one with the smallest magnitude AP is queued before the other.

Otherwise, namely if APi and APj differ in sign, the one of positive AP is queued before the other.

In IGS, daemons are ordered as stated above and the first daemon on the queue is popped off and executed until the queue is empty.

In JS, daemons are also be ordered in the manner stated above; the "or both negative" clause, however, can be removed from the second statement of (3).

JS differs from IGS primarily in the manner in which daemons are removed from the queue and executed. The action taken by JS when the next daemon is to be executed depends on whether the first daemon in the queue is a loop daemon, hence having a negative acyclic priority, or a non-loop daemon, hence having a positive acyclic priority. The two cases are handled as follows:

(1) If the first daemon on the queue has a positive acyclic priority, it is simply removed from the queue and executed as in IGS.

(2) If the first daemon on the queue has a negative acyclic priority, the queue is scanned from its start, placing daemons passed

over on a separate list, until a daemon is found which either (a) has a positive acyclic priority, or (b) has a cyclic priority different from the first daemon on the queue; given the queueing order defined above, the second test actually subsumes the first. The daemons placed in that list are not yet removed from the queue. Then each daemon in the separate list is first removed from the queue and then run, in any order. When all the daemons in the separate list have been run, the first daemon on the queue is again examined and (1) or (2) ensues.

The fact that loop daemons are queued after non-loop daemons establishes, with action (1) above, Rule 5J, feedforward. Action (2) corresponds to Rule 6J, feedback. Since only daemons with the same cyclic priority are run in action (2), they must be in the same MCS and can be run in any order according to Rule 3, efficiency. Loop daemons are not removed from the queue until after they are run in order to follow Rule 1, selection: a loop daemon could OUCH an output watched by a another loop daemon which has yet to be run in action (2).

Once the two priorities are established, the process of queueing daemons and running them entails relatively little overhead. This is an important characteristic, since smooth motion usually entails a great many re-executions of an M-DALI data web with relatively little structural change.

However, it is equally clear that the job of establishing cyclic priorities is fairly expensive, due to the necessity of constructing or maintaining the interconnection matrix C. We can, however, limit the number of times C this must be constructed or modified to these two:

(1) a daemon is created which specifies an output already watched by a loop daemon

(2) any structural change is made to the data web by DEPENDS, etc.

If either of these happens, cyclic priorities must be re-established, and, in the second case, acyclic priorities must also be re-established. This must be done as soon as the current daemon is

finished executing. After priorities are re-established, the daemon queue will have to be re-ordered, and finally execution of daemons can be continued.

Simple, unlooped creation of daemons can be handled by giving the new daemon a cyclic priority greater than that of any of its ancestors, the way acyclic priorities are initially established. Deletion poses no problem, even if cycles are broken, thanks to acyclic compatibility: the scheduling information conveyed by the two priorities may become redundant or overly restrictive, but it is never wrong; i.e., even if every loop daemon in a cyclic data web is deleted, daemons will still be queued in the right order.

So the situation is, perhaps, not as bad as it might appear at first glance.

5.9 Implementing Multi-Way Constraints: SKETCHPAD Revisited

Daemons, loop or normal, most closely resemble one-way constraints. Although there are cases where daemons can be successfully used to directly model apparent multi-way constraints, as used in SKETCHPAD and described in section 1.2, many such cases are doomed to non-convergence.

For example, the classic case of relaxing several values to be equidistant from two neighbor values can be directly modelled. In this case, there are n+1 values, namely A(0) through A(n); and each A(i), for i in the range $0<i<n$, "tries" to keep itself midway between its neighbor values A(i-1) and A(i+1). This can be expressed, effectively using the equation form of the data web, as follows:

A(0)=externally specified
A(1)= (A(0)+A(2))/2
.
.
.
A(i)=(A(i-1)+A(i+1))/2
.
.
.
A(n-1)=(A(n-1)+A(n))/2
A(n)=externally specified

All that is needed is n-2 identical loop daemons, each of which specifies one of the output values A(1) through A(n-1), watches its two neighbors, and performs the obvious calculation and OUCH each time either of its neighbors changes. Convergence tests are also needed. The data web for this is shown in Fig. 5-6 for n=6.

However, as Sutherland points out in [Sut1], cases like the following cannot be modelled in such a simplistic fashion:

```
A=2*B
B=2*A
```

If this is done with two daemons -- one watching B and forever OUCHing A to 2*B, the other watching A and forever OUCHing B to 2*A -- the system will diverge for any non-zero initial values of A and B. This data web is shown in Fig. 5-7. In comparison, a least-mean-square error relaxation such as that of SKETCHPAD will converge to A=B=0, the stable result.

However, this does not mean that multi-way constraints cannot be modelled at all. The problem is simply that the daemons are not doing the calculation required to model both of the constraints. For least-mean-square error relaxation, each daemon should utilize the two error terms needed to describe both constraints simultaneously, namely

$$e1=A-2b \quad \text{and} \quad e2=2B-A\ .$$

To find what the daemon for A should do, we must take the sum of the squares of the error, then take the derivative of that with respect to A and equate it to 0 to find the minimum. Solving that equation for A yields

$$A=B*4/5$$

which is the calculation to be done each time that daemon is called. Similarly, the B daemon should compute B=A*4/5. One or the other, of course, needs a convergence test.

The above technique is not directly applicable to the general case, since a different daemon would be needed for every possible combination of constraints. For the general case, the solution should be split

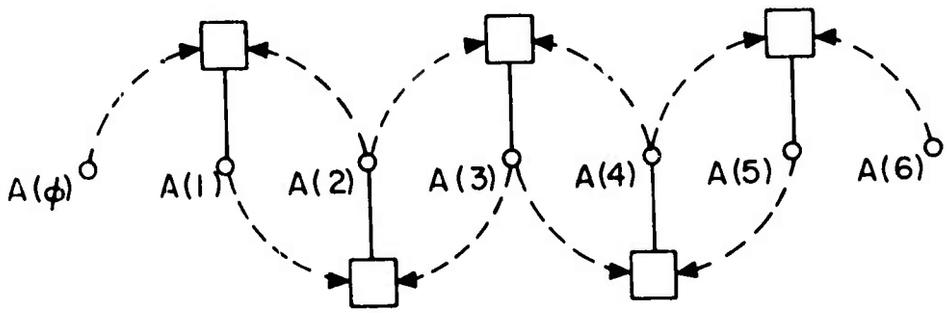


FIGURE 5-6 TWO-WAY CONSTRAINTS DIRECTLY MODELLED

A = 2*B
B = 2*A

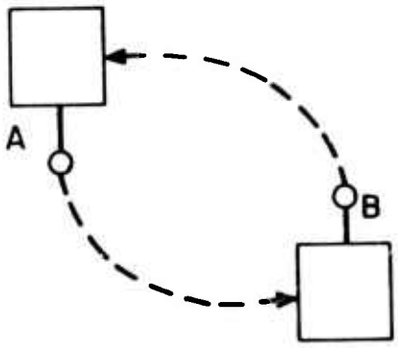


FIGURE 5-7 TWO-WAY CONSTRAINTS INCORRECTLY MODELLED

between two classes of daemons, called arbiter daemons and constraint daemons:

> There is one arbiter daemon specifying each constrained output; its job is to find a value for that output which is least distasteful to the constraint daemons specifying outputs he watches.
>
> There is one constraint daemon per constraint and per output constrained. These specify outputs which tell the arbiter for the constrained output how to satisfy the constraint daemon's particular constraint.

For example, suppose the method of solution chosen is least-mean-square fit to linearized constraints, as in SKETCHPAD. Then each constraint daemon would feed an arbiter the coefficients m and b of the linearized error term

m*V+b

to constrain the value of the output V. For the two constraints

(1) A=2*B

(2) B=2*A

and initial values A=1 and B=2, the arbiter for A would be given

(1) m=1, b=-4

(2) m=-2, b=1

by constraints (1) and (2) respectively. Solving this, in a manner to be shown, yields a new A of 1.6. Then B's arbiter would work on

(1) m=2, b=-1.6

(2) m=-1, b=-3.8

yielding a new B value of 1.28, etc. The data web evolving out of this approach is shown in Fig. 5-8, which illustrates two general two-way constraints constraining two values. The daemons labelled 1 are constraint daemons for constraint 1, and those labelled 2 impose constraint 2. The two loop daemons labelled arb are arbiters.

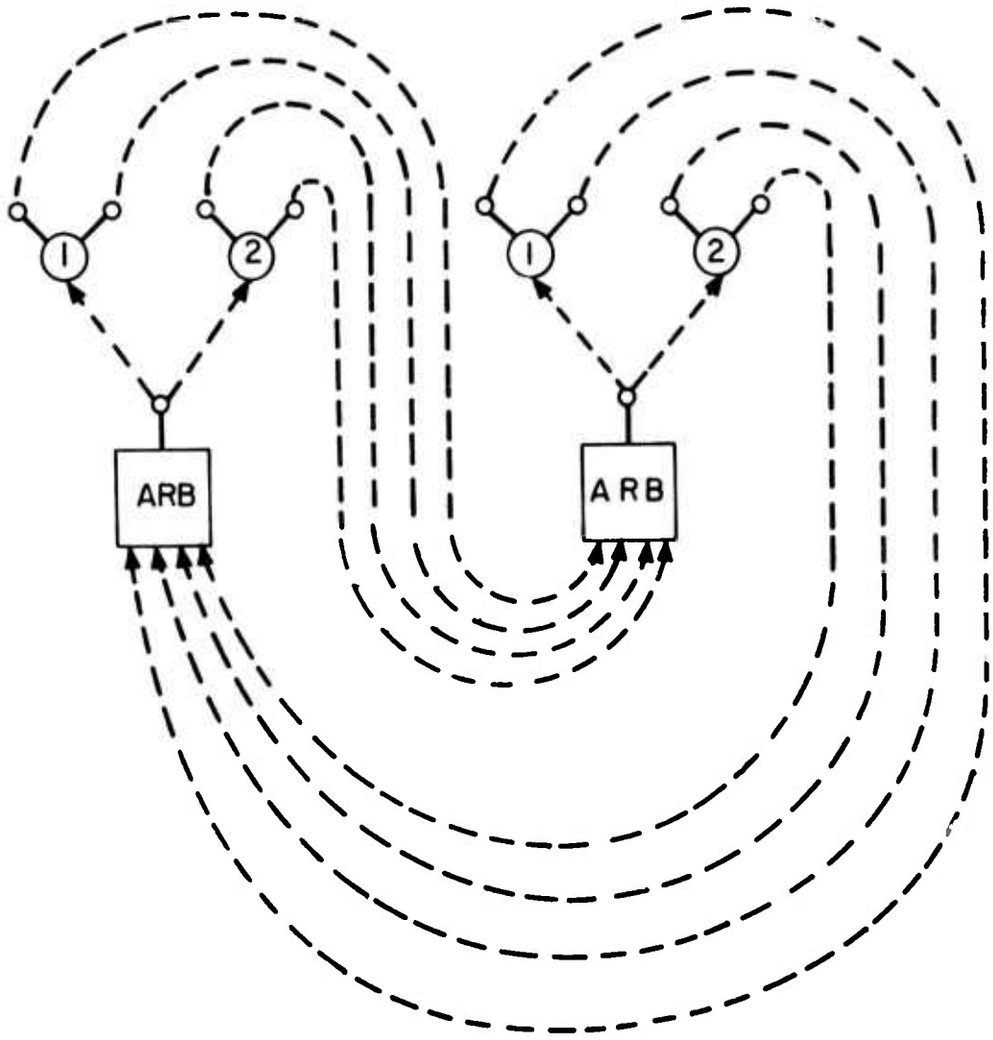


FIGURE 5-8 GENERAL DATA WEB FOR TWO VALUES CONSTRAINED BY TWO-WAY CONSTRAINTS

A general linearized-least-mean-squared error arbiter can easily be constructed as a picture module containing a daemon watching a variable number of m and b error terms. His job is to select a value V minimizing the sum over i of

$$[m(i)*V+b(i)]**2$$

which, taking the derivative with respect to V and equating the result with 0, means OUCHing V to

$$\frac{-m(i)*b(i)}{m(i)**2}$$

where the numerator and denominator are separately summed over i.

A picture function for a least-mean-squared error arbiter is LMS, below:

```
(DEFPIC LMS (EPS "AUX" MBLIST DEM "OUT" V "OUTU" ADDIT)
  (SETQ DEM
    (LOOPON (VAL EPS) (V)
      (PROG (NEW)
          (SETQ NEW (SOLVE MBLIST))
          (COND ((> (ABS (- ,V NEW)) ,EPS)
                (OUCH V NEW)) ))))
  (SETQ LMS (SETQ LB NIL))
  (SETQ ADDIT (P-CLOSURE ADD-CONSTRAINT)) )
```

In LMS, V is the output whose value is to be constrained; MBLIST is a list of (m b) pairs, and the p-closure assigned to ADDIT is used to add constraints. EPS is an output defining the allowable error, probably shared across a section of the data web. OUCHing EPS to a lower value will cause that section of the data web to "tighten up" its constraints.

SOLVE, which solves for the least-mean-squared error value of V, is:

```
(DEFINE SOLVE (MBLIST)
  (PROG (NUM DEN)
    (SETQ NUM (SETQ DEN 0))
    (MAPC
      (LAMBDA (MB) (SETQ NUM (+ NUM (* ,(CAR MB) ,(CADR MB))))
                   (SETQ DEN (+ DEN (* ,(CAR MB) ,(CAR MB)))))
      MBLIST)
    (RETURN (/ (- NUM) DEN)) ))
```

While ADD-CONSTRAINT, which is used as a p-closure relative to LMS' local environment, is:

```
(DEFINE ADD-CONSTRAINT (M B)
    (WATCHES DEM M)
    (WATCHES DEM B)
    (SETQ MBLIST (CONS (LIST M B) MBLIST))
    (ON-DELETION M KILLIT)
    (ON-DELETION B KILLIT)
    (RUN DEM)
    T)

(DEFINE KILLIT (DEADOUT)
   (PROG (MB)
     (SETQ MB (FIND-IN-MBLIST DEADOUT))
     (COND (MB (UNWATCH DEM (CAR MB))
               (UNWATCH DEM (CDR MB)) ))))
```

Basically, ADD-CONSTRAINT adds a new (m b) pair to MBLIST. It also makes sure that deletion of the constraint does not delete the arbiter, by putting onto M and B a deletion p-closure using KILLIT. KILLIT uses FIND-IN-MBLIST to find the (m b) pair in MBLIST containing the dead output. FIND-IN-MBLIST returns the (m b) pair after splicing it out, if such a pair exists; otherwise it returns NIL (=false).

For the example case of A=2*B, B=2*A, a picture module applying the constraint A=2B to two values arbited by modules X and Y would be:

```
(DEFPIC X2Y (X Y "AUXO" B1)
   (ONS (VAL (OUT Y))
      (OUCH B1 (* ,(OUT Y) -2)))
   (APPLY (OUT X 2) (NULLSPEC (OUTPUT 1)) B1)
   (APPLY (OUT Y 2) (NULLSPEC (OUTPUT -2)) (OUT X)) )
```

Thereby providing X with the error X-2Y, or m=1, b=-2*y; and Y with the error -2*Y+X, or m=-2, B=X. Deleting such a constraint module will remove the constraint gracefully, thanks to the ON-DELETIONs of ADD-CONSTRAINT.

The arbiter and constraint-daemon method of implementing multi-way constraints is both more flexible and more efficient than the multi-way constraint mechanism of SKETCHPAD.

It is more efficient than SKETCHPAD in this sense: When SKETCHPAD discovers cyclic constraint dependence, it relaxes the <u>entire</u> system on each iteration, right down to the visible line positions. No attempt is made to isolate sections of the picture which needs relaxation from

sections which do not. This contrasts with DALI's efficiency criterion in cyclic scheduling, which guarantees that only those portions of the data web needing relaxation actually participate in the relaxation.

DALI's method of implementing multi-way constraints is more flexible because arbiters can be written to accommodate a wide variety of approximations to constraints. For example, arbiters of second-order approximations -- a(1)*V**2 + a(2)*V + a(3) -- can be written. More generally, arbiters can be viewed as objects providing approximate, iterating solutions to situations with multiple simultaneous goals. It might even be possible to design arbiters which operate on qualitative, rather than quantitative, descriptions of error. For example, an arbiter might handle an "error comment" from a constraint daemon of the form "This position value will be OK if it's below that line, but it would be better to have the value in this triangular region; however, make sure you keep it near that piece of text, and whatever you do, don't put it in the upper right-hand corner!"

Chapter 6
S-DALI: Interpolated "Smooth" Change

"Time is a system for keeping everything from happening at once."
--Anonymous
(found on the wall of the Orson Wells Cinema in Cambridge, Mass.)

6.1 Motivation

The general purpose of S-DALI is to extend the capabilities of M-DALI to the handling of sequences of monadic changes across viewer-perceived picture time. By doing so, S-DALI encompasses the generation of "smooth" interpolated motion in a general way. This section describes and motivates the principal features of S-DALI.

It should initially be recalled from section 2.3 that picture time is time as it is intended to be perceived by the viewer, as opposed to the DALI compute time needed to compute individual "frames" and inter-"frame" relationships.

In the context of M-DALI, change which is to be perceived by the viewer as "smooth" is best provided by creating picture-temporal sequences of monadic changes to an output. The illusion of continuous motion can then be achieved by appropriately choosing sequence elements, e.g., by interpolation. In this way the normal operation of M-DALI, repeated for each element of the sequence(s) in progress, can keep relationships between picture elements correct during such change.

Thus, for example, a "smoothly moving" visible line can be created by applying appropriate sequences of monadic changes to the inputs of the LINE picture module defined in the discussion of M-DALI.

S-DALI provides such a capability; but this is not the full purpose of S-DALI.

S-DALI also allows computation to be performed on entire sequences of changes in the way that M-DALI allows computation to be performed on individual instantaneous changes: entire sequences can be passed through the data web, operated on as entities, and propagated further. I.e.: S-DALI is to sequences as M-DALI is to instantaneous values.

This is more precisely expressed by saying that S-DALI allows daemons to generate temporal sequences of monadic changes to outputs such that:

(1) generated sequences can be functions of other determining sequences taken as wholes, not just functions of other sequences' elements; and

(2) both generated and determining sequences can appear temporally simultaneous to the viewer.

This capability is provided by a new class of daemon conditions which "watch for" sequences applied to outputs. Daemons with these conditions are run before the occurrence of any of the value changes represented by the applied sequences. Thus these daemons can inspect entire sequences, using them as data to compute new sequences whose changes will occur in parallel with the data sequences. Applying such new sequences to outputs can then propagate sequence information through a data web just as value changes propagate in M-DALI. All the previously created sequences "really" happen, i.e., the picture times associated with their elements occur, after such sequence processing has taken place.

The S-DALI data web used for propagating sequence information is not intrinsically separate from the M-DALI data web; but the manner in which "sequence watching" daemons are queued relative to "value

watching" daemons causes interaction between them to be easily comprehended and used.

That sequences can be functions of other simultaneous sequences, i.e., sequence -> sequence functionality exists, is the characteristic that distinguishes S-DALI from an M-DALI system with a looping driving program; this characteristic also distinguishes S-DALI from simulation systems such as that embedded in SIMULA [Dah1, Dah2]. The motivation behind, first, sequence -> sequence functionality, and second, the need for simultaneity, will now be discussed.

Without sequence -> sequence functionality, certain classes of behavior are impossible to produce within the picture itself, i.e., within picture modules.

A simple example demonstrating this is illustrated in Fig. 6-1. There, outputs O1 and O2 specify positions, and O2 is a function of O1. O2 is to be a vector distance D from O1 when neither are moving. When O1 moves, O2 is to move in a semi-circular path to its final position, as shown. This cannot be done unless information about O1's final position is available to O2 before O2 starts to move: O2's initial "step" is dependent on O1's final position, since that position determines the radius of O2's semicircular path.

A slightly more complex example is shown in Fig. 6-2; here O1 and O2 are again positions, and O2 again depends on O1. When O1 moves, O2 is to end up in the same position as O1, moving there in a series of "hops" normally of length S. If O2 is to land in exactly the right position, the final "hop" must be made smaller than S, as shown; thus the final position must be known in order to determine the full configuration of "hops". The type of cyclic motion shown in this example is characteristic of walking figures; final, and, for walking, initial motions differ from intermediate motions because standing posture differs from posture while walking. Clearly, the final value of the determining sequence -- here O1's -- must be known before a cycle can commence.

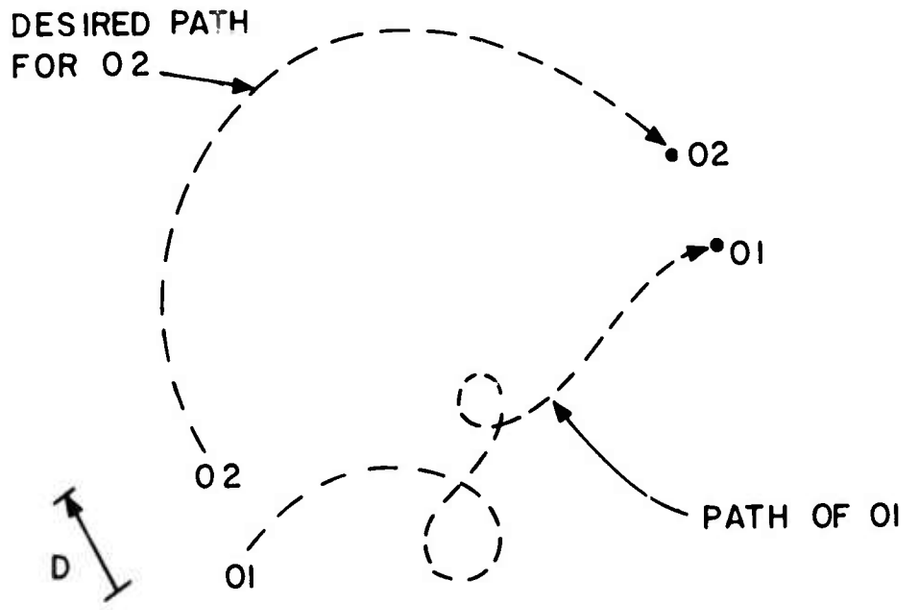


FIGURE 6-1

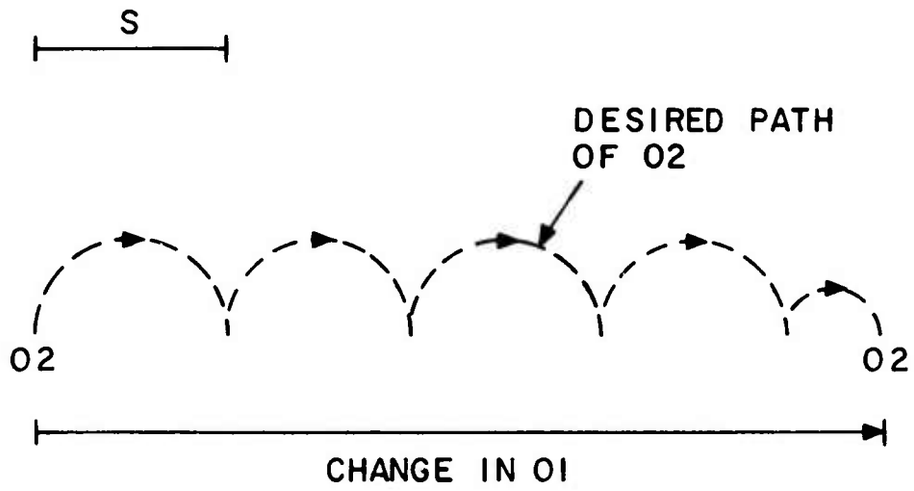


FIGURE 6-2

The requirement that computed sequences be able to occur simultaneously in picture time with their determining sequences is justified by the fact that sequence -> sequence functionality can be nested to any depth; thus any visible delay caused by such nesting could result in arbitrarily bad loss of synchronization. This could in theory be compensated for by clever programming, but only at an immense loss of modularity.

An example showing how such nesting can become rather deep, and also giving the intended "flavor" of S-DALI programming, is the action of a (hypothetical) program for a human figure as it catches a thrown ball:

A top-level "human" daemon is passed, through an output, an object such as this two-element list:

(CATCH ball-s)

where ball-s is the sequence specifying the ball's motion. Seeing the CATCH token, this top-level daemon passes ball-s through an output to a "ball catcher" daemon, which examines the ball's sequence and issues a sequence specifying an appropriate hand position and time for a catch. This causes a "major skeletal coordination" daemon to note, for example, that the position is near enough that running and jumping is unnecessary; so it just issues a set of sequences specifying appropriate shoulder, elbow, and wrist motions. A specialized "hand control" daemon, noting the wrist and hand position sequences, issues a set of sequences to orient fingers and palm appropriately and close them at the proper time for the "catch"; this, in turn, might set off more specialized daemons, e.g., for thumb movement. Finally, a set of "skin and musculature" daemons reacts to all the skeletal sequences with appropriately realistic bulges, throbs, and wrinkles in the visible covering of the (invisible) skeleton [Mon1]. Only after all this is done in DALI compute time does anything actually start moving in picture time.

All of the above will produce a very strange picture indeed if

there is a visible picture-time delay inherent in sequence creation: not only will the hand reach its place after the ball -- which is a much simpler object with less nesting -- but the hand will lag after the forearm, and who knows when the thumb will get there?

Similarly complex cases can be found in seemingly simpler operations, such as moving pointers about in a display of the workings of an interpreter or compiler; only the skin is missing. The above example just illustrates the problem more graphically, and, besides, it's more fun.

If sequence -> sequence functionality is to create sequences which are to occur in parallel with their determining sequences, knowledge of the future is needed -- i.e., the values which an output <u>will</u> take on. This information is useful in two ways.

First, the availability of future data can reduce the total amount of computation needed by reducing the need to calculate predictions. For instance, in the "catch" example above, the "catcher" daemon could have been written in the following fashion: it could wait, recording some successive ball positions, and then calculate the trajectory it needs to know. With access to the ball's future position, however, at least part of that calculation can be avoided. The case is even clearer when many objects are "watching" a particular object. The moral of this seems to be that the more of the future an object knows, the less intelligent it need be.

Second, there is another interpretation of such sequences: determining sequences are <u>orders</u>, i.e., partial specifications of future behavior which an object (a picture module) is to satisfy by its own methods. Here, the time element of a determining sequence is considered to be a part of the order, specifying when the behavior is to be carried out.

Thus, in the two simple sequence -> sequence examples above, the sequence applied to 01 was interpreted as an order to move 02 to a new

position, accomplishing this action while O1 was moving; the O2 code "decided" to do so by different means in each case.

The stratagem of allowing the response to such orders to be the same type of object as the orders themselves -- sequences -- creates a desirable uniformity that allows orders to propagate through a picture in exactly the same way that monadic changes propagate in M-DALI. Making such orders interpretable as M-DALI operations smooths their transmittal to the "cannon fodder" M-DALI routines which perform operations more easily done without viewing the future.

The requirements of sequence -> sequence functionality and simultaneity put a significant restriction on how the elements of sequences may be generated when they are "really" applied to outputs: all the elements of a sequence must be computable before the picture time of the first element is reached. Thus a sequence generator must be a function of one variable, namely picture time: it can have no internal state. I.e., the apparently "natural" model of sequences, general finite-state machines, is too general; sequences can at best be modelled by information-lossless finite-state machines which can always be "backed up" in time. This is truly a significant restriction, as it disallows many elegant, general schemes which effectively use co-routines as generators of sequences; such schemes are, for example, used in PLANNER, CONNIVER, and SIMULA 67 [Hew1, McD1, Dah1, Dah2].

To some extent this restriction will be relaxed. For example, in the "catcher" example above the trajectory of the ball may be initially computed as if no catch were to be made; and a later modification can be made when the catch is imminent, appropriately decelerating the ball. However, the nature of the facilities of S-DALI militates against general sequence generators.

There are, nevertheless, cases where suitably complex sequences are used in situations where only M-DALI daemons need watch the outputs they change. Since the facilities of S-DALI are not being used in these

cases, "value generators" with state can be used, and are allowed. They do not, however, constitute or generate true S-DALI sequences, and in particular cannot excite "sequence watching" daemons as do true sequences.

6.2 S-DALI Operation and Action Scheduling

The operation of S-DALI is similar to that of the event-driven simulation system embedded in SIMULA [Dah1]. It is based on an agenda, which is a linked link of agenda blocks (blocks), each containing two elements:

(1) a picture time

(2) a set of actions

There must also be a "next agenda block" entry to define the list; but in this discussion it is irrelevant.

The picture time indicates when the set of actions will be performed. The agenda is continuously kept sorted in order of increasing agenda block picture time.

The set of actions essentially contains blocks of executable code -- actions -- which are to be performed at the block's time. Typical actions include OUCHing an output, deleting an object, and starting up a sequence. The set of actions has internal structure which defines a partial order in which actions are performed; these topics will be discussed in section 6.5.

S-DALI's total operation consists of removing the first block from the agenda and then processing it in a manner to be described, repeating this performance until the agenda is empty. This operation is initiated when the driving program performs an action which causes the generation of an agenda block -- for example, by directly applying a sequence of

changes to an output, or by doing an OUCH which causes a daemon to apply such a sequence, and then performing an UPDATE-DISPLAY. S-DALI operation, like M-DALI operation, occurs nested within the UPDATE-DISPLAY; when the agenda is empty, control again returns to the driving program.

The agenda block which is currently being processed is referred to as the current agenda block (current block), and its picture time is called the current time. A user can obtain the current time as the value of the function CURT applied to no arguments.

Agenda blocks are created by an operation called action scheduling (scheduling). To schedule an action A for a picture time T,

(1) If T is the current time, do not actually schedule A; instead perform the action immediately exactly as it would be performed in normal block processing.

(2) Search the agenda for a block with picture time = T. If such a block exists, insert A into the block's set of actions.

(3) If no such block exists, create a new block with picture time T and put A into its action set. Insert the new block into the agenda immediately before that block with the smallest time greater than T; if no such block exists, put the new block at the end of the agenda.

Rule (1) is not just a trick to avoid unnecessary agenda block creation; it is necessary for "sequence watching" daemons and data web to work correctly, since such daemons watch only for actions which are scheduled. Furthermore, it cannot be combined with Rule (2) by leaving the current block on the agenda. This will be explained later in this section.

Processing of the current block is performed in two sequential phases, the first called the initiation phase and the second called the follow-up phase:

initiation: the set of actions is performed (in an only partly defined order). If there are daemons watching for any of these actions, place them on the daemon queue according to the (M-DALI) daemon scheduling rules. If deletion is performed during this phase, actual interment is deferred until the start of follow-up.

follow-up: Run the queued daemons in accordance with the (M-DALI) daemon scheduling rules.

The division into two phases is what necessitates scheduling rule (1). This rule causes actions which are scheduled for the current time to be performed immediately, thus possibly performing what otherwise would be initiation phase activities during in the follow-up phase.

Three things distinguish current block processing from the driving program/M-DALI interaction described earlier: a wider variety of actions can be performed, a new class of daemon conditions can be invoked, and one anti-data-web-cycle prohibition -- that only an output's specifier may OUCH an output -- does not operate during the initiation phase.

New actions include: action scheduling, which invokes no daemons; initiating a sequence, which invokes daemons with the new class of conditions; and continuing a sequence -- performing an OUCH and re-scheduling the continuation -- which invokes normal M-DALI daemons in response to the OUCH. These are discussed in detail in a following section.

Daemons having the new class of conditions, referred to as S-DALI daemons as opposed to M-DALI daemons, are distinguished by the fact that all M-DALI daemons are queued before any S-DALI daemon. This is clearly not the mechanism by which processing of time sequences is made to precede the processing of monadic events. Among the S-DALI daemons the order of queuing is determined by ancestry in the data web, using the same scheduling rules as M-DALI daemons.

During the initiation phase, the set of actions performed can include OUCHing or applying a sequence to any output whatsoever,

independent of whether the output has a specifier or what that specifier is. If an output is OUCHed more than once, its value remains that given by the last performed OUCH -- recalling that the order in which initiation phase actions are performed is not completely defined. The reason for allowing such uncontrolled behavior is that the primary reason for disallowing it does not exist. Specifiers exist to make a true representation of inter-module functional dependence available to the scheduling rules so that they can operate with rationality and approbrium in propagating change; but change is not being propagated during the initiation phase. Rather, change is being inserted into the picture definition for propagation during the follow-up phase, and in that phase all the usual restrictions apply. This is the source of the "future OUCH" mechanism mentioned in sections 4.6, 5.2 and 2.3.

If initiation phase activities are performed during the follow-up phase due to action scheduling rule (1), they <u>are</u> subject to the anti-circularity "specifier" rule which is suspended during the initiation phase; this suspension is associated with the phase, not with the actions performed.

6.3 General Scheduled Actions

A straightforward method of causing an action to be explicitly scheduled for a given time is to create a <u>scheduled p-closed action (SPA)</u> by means of the SCHEDULE function. Doing so does not create a true sequence; that operation is covered in the next section.

An SPA is always a member of some agenda block's set of actions, and the action it describes is performed during the initiation phase of agenda block processing, except, as mentioned, when it is scheduled for the current time.

A scheduled p-closed action has three elements:

(1) the owner -- a picture module

(2) the body -- an s-expression (block of code)

(3) the agenda block

In addition, since an SPA is deletable it has the standard three deletion elements.

The owner of an SPA is the owner of the daemon which created it, the body is an arbitrary block of code, and the agenda block is that block whose set of actions includes the SPA.

An SPA is performed during the initiation phase of its agenda block's processing as follows:

(1) It is evaluated as if it were one of its owner's daemons.

(2) If it has not re-scheduled itself, it is deleted.

Re-scheduling is covered below; the test in (2) is quite simple.

A scheduled p-closed action is created and initially scheduled with the SCHEDULE function:

(SCHEDULE ti -body-)

where ti is a number which is added to the current time to obtain the SPA's scheduling time, and -body-, an arbitrary list of S-expressions, becomes the body. The agenda block is obtained in the process of scheduling, which occurs as part of SCHEDULE. SCHEDULE returns the new SPA as a value. If SCHEDULE is called with ti=0, the -body- is immediately executed; this is useful for initialization, and follows from the action scheduling rules.

An already scheduled SPA can be re-scheduled for a different time

(RE-SCHEDULE ti spa)

where spa is the SPA, and ti is again added to the current time to obtain the time for (re-)scheduling. RE-SCHEDULE causes the SPA to be removed from the set of actions of its agenda block and scheduled again, changing the SPA's agenda block appropriately. The spa argument is optional and defaults to the currently running SPA; thus

(RE-SCHEDULE ti)

re-schedules the SPA which executes it.

The test for whether an SPA has been re-scheduled, needed in the processing of the SPA, is: if the SPA's agenda block is the current block, it has not been re-scheduled. RE-SCHEDULEing for the current time produces, due to infamous scheduling rule (1), an odd form of iteration which has no intrinsic benefit; therefore, as a safety measure, a 0 ti argument to RE-SCHEDULE is illegal.

A simple example:

Assume that MOVER and DELT are identifiers in the environment of the module owning the daemon containing the following application, and that MOVER is bound to an output. Then:

```
(SCHEDULE 0 (OUCH MOVER (+ ,MOVER DELT))
            (RE-SCHEDULE 10) )
```

creates an SPA which causes MOVER's output value be be incremented every 10 time units by whatever happens to be in DELT at the time, starting immediately. Note that this will go on until the SPA is deleted, or until its owner module is deleted. The latter occurs because when a module is deleted, all the objects owned by the module deleted, including SPAs.

6.4 Sequences and Simple Sequences

A sequence is an object representing a temporal sequence of monadic value changes which a single output will undergo. Each such change is completely equivalent to an OUCH, and will cause daemons watching for value changes to be executed. Like a scheduled p-closed action, a sequence is always a member of some agenda block's set of actions, and only does something when it is performed during the initiation phase of block processing. Sequences are accessed and examined via the outputs

they control, and only one sequence can control an output at any given virtual time. After their creation, sequences cannot be altered. They can, however either be explicitly DELETEd or replaced by other sequences and thereby automatically DELETEd.

This section discusses simple sequences. These are sequences containing an explicit ordered set of values to be assumed by an output. More complex sequences -- path sequences, discussed in a later section -- simply substitute a function for this ordered set, and are otherwise identical.

The elements of a simple sequence are:

(1) the changee -- an output

(2) the value set -- an ordered set of objects

(3) the step index -- a positive integer

(4) the step duration -- a real number

(5) the agenda block

Simple sequences also have the standard three deletion elements because they are deletable.

The changee is that output which will be changed by this sequence.

The value set specifies the values the changee will assume, and the order in which they will be assumed. It must have at least one element.

The step index is incremented by one each time the sequence is processed. It counts from 0 to the length of the value set, and is used to tell both which element of the value set is next and when the sequence is finished. A sequence initially has a step index of 0.

The step duration is the picture time interval between the individual monadic changes.

The agenda block is the block whose action set contains the sequence.

Simple sequences are created by means of the function SOUCH (for Simple sequence of OUCHes), as in

(SOUCH out vset ftime stime)

where: out is the output which becomes the changee; vset is a list of values, which becomes the value set; ftime and stime are picture times specifying, respectively, the finishing and starting picture times of the sequence. stime is optional, and defaults to the current time.

SOUCH creates a new simple sequence, schedules it for stime, and returns the sequence as the value of the application. If an uninitiated sequence with the same changee output is already scheduled for the same time, the old sequence is DELETEd. What happens when a new sequence overlaps an old one without starting at the same time will be covered below.

The step duration is

$$\frac{\text{(finish time) - (start time)}}{\text{(length of value set)}}$$

Thus the picture time tm(i) of each monadic change i, where i ranges between 1 and the length of the value set inclusive, is:

$$tm(i) = \text{(start time)} + i*\text{(step duration)}$$

This is illustrated in Fig. 6-3. A sequence will be scheduled as continuing for each tm(i); this is described below.

Note that no value change occurs at the start time; this is if successive sequences applied to the same output are to dovetail appropriately. However, S-DALI daemons watching the changee are run at start time; this is described in the next section.

6.5 The Starting and Continuation of Sequences

In describing the processing undergone by sequences during the initiation phase of block processing, two other facts become relevant:

First, outputs gain two more elements in S-DALI: They now have a current sequence element, containing that continuing sequence currently

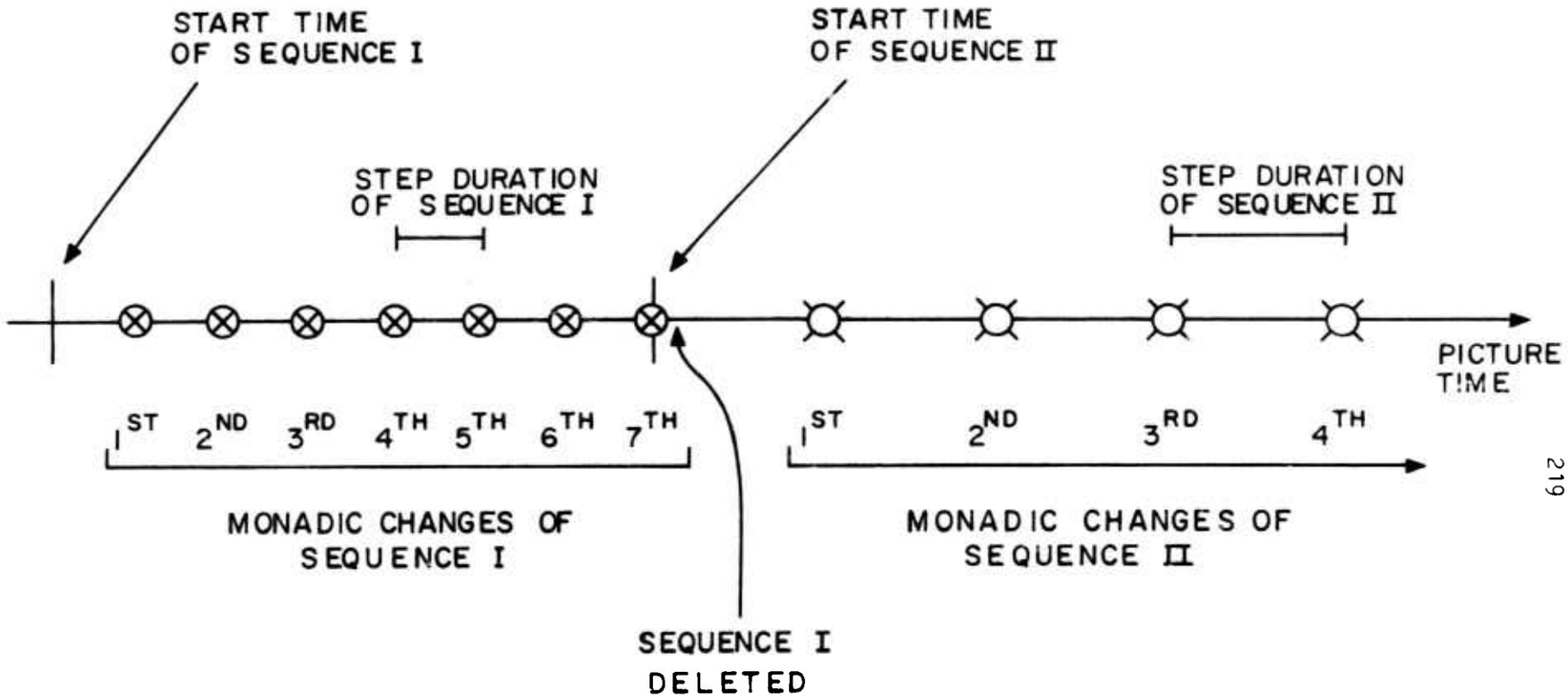


FIGURE 6-3 OVERLAPPING SEQUENCES

controlling the output. If no such sequence exists, this element contains a null value. If a sequence being DELETEd is the current sequence of its changee when interment rolls around, the changee's current sequence is made null. The other new element is a list of dependent S-DALI daemons; these are the daemons whose conditions "watch for" sequences applied to the output. How such daemons are created will be discussed in the next section.

Second, the set of actions of an agenda block is divided into 3 parts:

(1) continuations -- re-scheduled continuing sequences

(2) starters -- sequences scheduled into this block which have not yet begun

(3) SPAs -- scheduled p-closed actions scheduled into this block.

The only reason for separating starters and SPAs is to ease the problem of finding and DELETEing sequences created by redundant SOUCHes.

Continuations are separated from starters so that they can be performed first, as described below.

The partial ordering of processing in the initiation phase of block processing is:

(1) First, the continuations are processed in an undefined order.

(2) Then, the starters and SPAs are processed in an undefined order.

Recall that this entire phase -- both of the steps above -- proceeds uninterrupted by daemon execution and interment. Daemons are queued, and interment is deferred, until the follow-up phase which follows initiation.

As implied above, sequence processing comes in two varieties: start processing and continuation processing. While for a given sequence start processing always comes first, any continuations in the current block's set of actions are done before starts.

For a sequence S, start processing consists of the following:

(1) If the changee of S has a current sequence, DELETE that sequence.

(2) Make S the current sequence of its changee.

(3) Queue the daemons with conditions watching for sequences on this output.

(4) Set S's step index to 1.

(5) Re-schedule S as a continuation for the current time plus the step duration.

This same start processing is performed on all sequences, simple or otherwise. The DELETE in step (1) defines the abjudication of overlapping sequences: the most recently started sequence always takes control of its changee.

Continuation processing of a simple sequence S proceeds as follows:

(1) OUCH S's changee to the (step index)th element of the value set.

(2) Increment S's step index by 1.

(3) If S's step index is greater than the length of its value set, DELETE S. Otherwise, re-schedule S as a continuation for the current time plus its time interval.

Agenda block processing produces an interesting and important sequence of occurrences when we find in the same agenda block (1) a continuation of an output's current sequence, and (2) an uninitiated sequence with that same output as a changee. The resultant behavior is:

(1) The continuation OUCHes the output to a new value. This causes the output's M-DALI daemons to be queued.

(2) The continuation is re-scheduled or DELETEd; either way, it won't last long, because

(3) The start processing of the new sequence DELETEs the continuation if it still exists.

(4) The new sequence becomes the output's current sequence, and its S-DALI daemons are queued; then the new sequence is re-scheduled for its first OUCH.

At this point both sets of daemons are queued, and the output has an "old" value and a "new" sequence.

(5) The M-DALI daemons are run.

(6) The S-DALI daemons are run.

Thus the value sequences dovetail correctly, including their computational descendants via M-DALI daemons. Also, the S-DALI daemons run at a picture time prior to that when the first change of a sequence takes place. Thus they run (1) before any M-DALI daemon sees the values from their driving sequences, and (2) in time to schedule sequences in parallel with their driving sequences.

6.6 Outputs and the SEQ Daemon Condition

In S-DALI, outputs are created in exactly the same way they are in M-DALI; however, they now also contain a current sequence and a set of dependent "S-DALI daemons".

An output's specifier is the only daemon which is allowed to either OUCH it or apply a sequence to it. Note, however, that any daemon may do either.

The current sequence is used to allow access to the sequence through the output. Functions used to do this are described near the end of this section.

The set of "S-DALI daemons" are the daemons watching for sequences applied to this output. They are queued, as previously described, when start processing is performed on the outputs they watch.

The only difference between an "S-DALI daemon" and any other daemon is its condition:

(SEQ -wouts-)

where -wouts- are outputs, is a daemon condition causing its daemon to be run whenever start processing is performed on any of the -wouts-.

In addition to ONC and ONS, a new daemon creating routine called ONCIF (ON Condition and IF changing) exists, which, like ONS, is a convenience for startup. ONCIF creates a daemon like ONC, but also runs its body before returning if any of the watched outputs is currently changing, i.e., any of them has a non-null current sequence.

The daemon ancestry relations on which the scheduling rules are based are slightly extended in S-DALI: a daemon with a SEQ condition is a web son of the specifiers of the outputs it watches. This means that "M-DALI" and "S-DALI" data webs can be intermingled. Usual program structure will normally keep them fairly separate, however, and, in any case, it makes no difference. The existence of M-DALI daemons between S-DALI daemons does not change the ancestry relations of the S-DALI daemons, and vice versa; and the separation of the running of S-DALI daemons and M-DALI daemons makes their relative ancestry irrelevant.

The scheduling rules of M-DALI are still used, modified by a new selection rule accounting for the SEQ condition, and extended to include the fact that all M-DALI daemons are run before any S-DALI daemon. This can be implemented either by having different daemon queues for M- and S-DALI daemons, or by adding a very large, fixed constant to the (inverse) priorities of all S-DALI daemons. The order in which S-DALI daemons are run relative to each other is defined by web ancestry using the rules of M-DALI. LOOPON can be used with a SEQ condition to create cycles in sequence processing.

Due to the difficulty of producing any reasonable examples using only SOUCH, examples are deferred until a sections 6.7 and 6.8.

As promised, the functions used to obtain data from sequences are listed below. These functions can be applied either to the sequences themselves or to outputs. In the latter case, information is obtained about the output's current sequence; this is usually the more convenient

method. In the following list, "os" is used to indicate "output or sequence":

(STEPS os)	total number of value steps this sequence takes
(STIME os)	start time
(FTIME os)	finish time
(IVAL os)	initial value
(FVAL os)	final value
(TVAL os ti)	value at time ti
(SVAL os i)	value at step number i
(CURSTEP os)	current value of step index

Three other functions also exist:

(SEQUENCE? out)

is a boolean returning "true" if and only if out has a current sequence.

(SEQUENCE out)

returns out's current sequence.

(CONTROLLED seq)

returns the output controlled by the sequence seq.

6.7 Path Sequences

The problem of describing arbitrary motion is a difficult one; it is equivalent to the problem of describing arbitrary curved lines and shapes, and is, strictly speaking, beyond the scope of the research presented here. However, SOUCH is clearly an inconvenient method for constructing sequences, and it is not difficult to do somewhat better. To that end, this section introduces the notion of a composite generalized path (path), a more flexible method of sequence specification than SOUCH. It is inspired in part by Baecker's "p-curves" [Bae1].

Paths are functions used as value generators in path sequences. A

path sequence is identical to a simple sequence except that the value set is replaced with two new elements:

(1) a step count

(2) a path

The step count is the number of monadic changes to be performed by the sequence.

The path is a function whose domain includes the real interval [0,1].

In the continuation processing of a path sequence, the changee output is OUCHed to the result of applying the path to the ratio

$$\frac{\text{step index}}{\text{step count}}$$

before the step index is incremented. If, after incrementing the step index, that index is greater than the step count, the sequence is DELETEd. Except for this, path sequences are processed exactly like simple sequences.

Path sequences can be directly created by the function NPOUCH, named for Naked Path-defined sequence of OUCHes, which is applied as

(NPOUCH out pth stepc fin strt) .

NPOUCH acts exactly like SOUCH, with the pair of pth and stepc -- a path and a step count -- replacing the list of values.

From an abstract point of view, NPOUCH is all that is necessary. However, it still suffers from being inconvenient. A better mechanism would be one which allowed the notion of shape to be specified independent of the notions of position, size, and rate of travel along the shape. To that end, the notion of composing a path will be pursued below.

The notion of "where in time", which conveys rate, is here represented by a time path. A time path tp is a function from the real interval [0,1] to that same interval, subject to the condition that tp(0)=0 and tp(1)=1.

"Shape" is represented by a shape path. A shape path sp is a function from the real interval [0,1] to any one of R, RxR, or RxRxR where R is the reals. A shape path has a dimensionality associated with the dimensionality of its range. To ideally separate rate and shape, sp should exhibit constant velocity: the distances travelled in its range during equal domain intervals are equal.

The composition sp(tp(t)), t in the interval [0,1], represents travelling along a shape, sp, with a possibly variable velocity defined by a distance function, tp. Such a composition will be called a gesture.

To allow such a composition to be "performed" in various positions, attitudes, and sizes, the concepts start point and end point are used. Basically, a gesture is translated, rotated, and scaled until sp(tp(0))=sp(0) coincides with the start point and sp(tp(1))=sp(1) coincides with the end point. The result of that transformation is the desired path.

Analytically, the composite path p(t) is a linear transformation of sp(tp(t)), determined by solving for the constants T and R in the equations

$$\text{start point} = p(0) = T + R*sp(0)$$

$$\text{finish point} = p(1) = T + R*sp(1)$$

The dimensions of sp, T, R, the start point, and the finish point must be compatible; for example, in a two-dimensional application everything would be a coordinate pair except R, which would be a 2x2 rotation and scaling matrix. If sp(0)=sp(1), the start point must equal the finish point; in this case T=0 and R is the identity matrix.

The final resultant path is used as the "path" element of a normal path sequence as described eariler in this section.

As an example, the use of the technique of cosine interpolation will be described. This technique can be used to make a motion gradually accelerate from rest and decelerate to a stop in a visually

pleasant way, and has the nice property that concatenated cosine interpolations produce "sine-curve-like" motion containing no visible discontinuities. In comparison, linear interpolation is visually ugly.

Cosine interpolation along a 1-dimensional straight line between any start and finish points can be done with:

```
sp(t) = t
tp(t) = 1 - cos(t)/2
```

Other dimensions are handled by making sp(t) be pos(t,t) or pos(t,t,t) as necessary; here pos(x,y) is the same as the LISP (POS x y).

Cosine interpolation along a counterclockwise half circle between any start and finish points can be produced with the tp above, and

```
sp(t) = pos(sin(3.1416*t),cos(3.1416*t))
```

The process of composing p(t) for this example is shown in Fig. 6-4.

In fact, cosine interpolation along any constant-velocity sp is produced with the given tp -- which is the whole point.

The above techniques, while fairly flexible, are still somewhat lacking. It would, for instance, be very useful to be able to supply extra parameters to sp and tp so that, for example,

```
sp(t,x) = pos(sin(x*t),cos(x*t))
```

could, by choosing an appropriate x, be used to move in arbitrary clockwise circular arcs.

An apparently reasonable way to do this is to allow sp and tp to be functional closures -- in this context, p-closures (section 4.3). Unfortunately, that would leave open the possibility of having the free variables in the closures change during the sequence. Not only does this violate the canon that sequences should be predictable, but it also implies that if the sequence is to end up at the right place -- the final point -- the T and R constants would have to be re-calculated at each monadic change.

An alternative is to use a "poor man's closure" -- i.e., allow the user to specify a set of constants to be applied to sp and tp in addition to the t argument. This is, of course, not a closure at all,

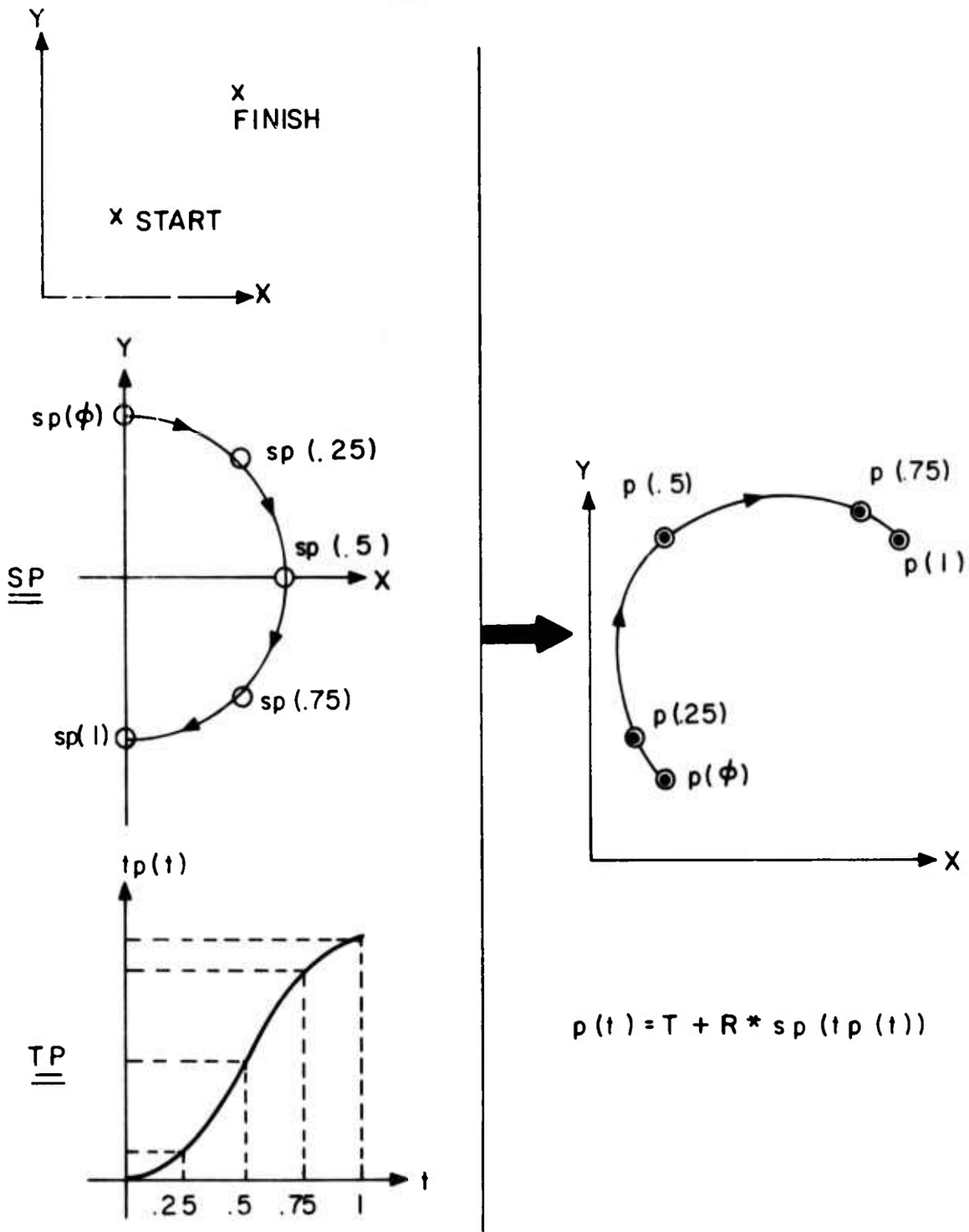


FIGURE 6-4 PATH COMPOSITION

but rather a convenient method of selecting a particular desired function out of a set of functions.

This latter alternative will be used, using a sample application as the method for specifying additional parameters. Such a specification then looks like

(fun ting -params-)

where fun is the time or shape path function, -params- are the parameters, and ting is any arbitrary place holder existing for mnemonic purposes only. When fun is applied, the first argument is always t, and the others are the given parameters in the order given.

Thus if we have

```
(DEFINE CIRC+ (T X)
        (POS (SIN (* T X)) (COS (* T X))) )
```

as a counterclockwise circular shape path, then a sample application

(CIRC+ "TIME" 4.712)

would result in motion around an arc of approximately 270 degrees.

At this point, the number of parameters needed to fully specify a path sequence has grown enormous. So, some simple syntax will be called to our rescue in the tradition of TRANSFORM and argument lists.

The function used to construct a path sequence and apply it to an output is MOVE. MOVE is applied as

(MOVE out -specs-)

where out is the affected output and -specs- is a sequence of designators and values used to specify the parameters of the path. The basic designators, the objects which follow each of them, and the quality specified are listed below. Each designator may occur only once, and the order of their occurrence is arbitrary. A designator in the list below which is preceded by an asterisk (*) is optional, and its default is given. "samp" is used as an abbreviation for "sample application".

*	"FROM"	value	initial value; default is current OVAL
	"TO"	value	final value
*	"SHAPE"	samp	shape path; default is a linear path
*	"TIME"	samp	time path; default is cosine interpolation
	"STEPS"	integer	number of steps
*	"START-TIME"	time	start time; default is current time
	"FINISH-TIME"	time	finish time

Thus,

```
(MOVE O5 "TO" (POS 100 100) "STEPS" 50
         "SHAPE" (CIRC+ "T" 3.1416)
         "FINISH-TIME" (+ (CURT) 100))
```

moves O5's value from wherever it is now to the position (100,100) along a cosine-interpolated semi-circular path in 50 steps, starting now and ending 100 time units from now.

In addition to the above, several convenience features exist to allow more compact specification of paths.

First, an output with a current sequence may be placed after any designator. If this is done, the corresponding element of the output's current sequence is used. For example, if "FROM" is followed by an output, the final value used is the final value of the given output's current sequence; similarly, "TO" followed by an output uses that output's final value.

Second, several other designators exist. They are listed below, along with their effects.

"FOR"	deltime	makes the finish time the current time plus deltime.
"RELTO"	delvalue	makes the final value the current OVAL plus delvalue.
"WITH"	out	makes the start time, finish time, and number of steps optional, defaulting to the corresponding parameter values of out's current sequence.
"FOLLOWING"	out	makes <u>all</u> parameters optional, defaulting to the corresponding parameter values of out's current sequence.

The primary difference between "WITH" and "FOLLOWING" is that when "WITH" is used, the standard defaults for "FROM", "SHAPE", and "TIME" are in force; whereas "FOLLOWING" changes these defaults. "DELTIME" and "RELTO" cannot be followed by outputs.

A third convenience measure will often be appropriate. It is often the case, particularly when movies are to be created, that the number of steps per unit time is fixed by the equivalent of a natural law to some standard frame rate. This also implies that virtual time has a natural quantization at the level of "frames". In this case, either the number of steps or the time duration of a sequence is a complete description of both duration and step count; hence either alone is sufficient. This will not be assumed in the examples which follow in the next section.

6.8 Examples

The first example is the simple one first mentioned in section 6.1 and illustrated in Fig. 6-1: motion along a semi-circular path, beginning and ending a distance D from the initial and final values assumed in the motion of a driving output. The code, which is expectedly simple, follows:

```
(DEFPIC SEMIMOVE (O1 D "OUT" O2)
  (OUCH O2 (+ ,O1 D))                    ;Set up initial value.
  (ONCIF (SEQ O2) (O1)
    (MOVE O2 "TO" (+ (FVAL O1) D)
             "SHAPE" (CIRC- "T" 3.1416)
             "WITH" O1) ))

(DEFINE CIRC- (T X)
   (POS (SIN (* (- T) X)) (COS (* (-T) X))))
```

The "WITH" causes O2's motion to occur in synchrony with O1's, while using the standard cosine interpolation time path default for O2's motion. A new shape path, CIRC-, was needed to produce the clockwise circle shown in Fig. 6-1; CIRC+, defined in the previous section, produces counterclockwise motion.

The next example, which is also simple, produces damped oscillatory motion in response to a step function input. In other words, when a "forcing input", FI, changes from some initial to some final value over a period of time, the damped output DAMPO will, in the same time period, move from FI's initial value out past FI's final value, swing back to less than FI's final value, swing forward but less than the first time, back again, etc., oscillating about FI's final value but converging to that value and stopping at exactly the right point at the end of the specified time period. The speed of convergence is an input parameter, DAMP, as is CYC, the number of cycles performed. The code for the DAMPER picture module producing this behavior is:

```
(DEFPIC DAMPER (FI CYC DAMP "OUT" DAMPO)
  (OUCH DAMPO FI)
  (ONCIF (SEQ FI) (DAMPO)
    (MOVE DAMPO "FOLLOWING" FI "TIME" (LINEAR "T")
                "SHAPE" (DAMPIT "T" CYC DAMP)) ))

(DEFINE DAMPIT (T CYC DAMP)
  (- 1 (* (EXP (- (* T DAMP)))  ;EXPonentiation to the base e.
          (COS (* T CYC 6.283)))))

(DEFINE LINEAR (T) T)
```

The MOVE of DAMPO "follows" FI, using FI's initial and final values and occuring in synchrony with FI's motion. The time path is made linear, since the DAMPIT shape path itself contains the velocity changes desired. DAMPIT itself simply implements an equation of a familiar form: exponentially damped oscillation.

DAMPER operates on a one-dimensional value. Its output could be used directly to specify a rotation, or indirectly to specify a size. Two coupled DAMPER outputs, one for X and one for Y, could be used to create damped two-dimensional motion which spirals in to the desired position.

Now, the second example of section 6.1 will be coded: an output moving from an initial position to a final position in a series of semi-circular hops. All but the last hop will cover a given constant length, HOPL, and the last hop will be truncated to land in the correct spot.

Intermediate hops will begin and end on a straight line connecting the initial and final two-dimensional values. The code for this picture function, HOPPER, follows.

```
(DEFPIC HOPPER (SO HOPL "OUT HOPO
                "AUX" HOPTIM HOPIX HOPDIST HOPSTEPS)
  (OUCH HOPO SO)
  (ONCIF (SEQ SO) (HOPO)
    (SETQ HOPIX (/ (DISTANCE (IVAL SO) (FVAL SO)) HOPL))
                                        ;Number of hops yet to be done.
    (SETQ HOPTIM (/ (- (FTIME SO) (STIME SO)) HOPIX))
                                                    ;Time per hop.
    (SETQ HOPDIST (* HOPL (ANGLPT (IVAL SO) (FVAL SO))))
                                ;Vector distance per hop. See text.
    (SETQ HOPSTEPS (/ (STEPS SO) HOPIX))
                                           ;Number of steps per hop.
    (SCHEDULE 0
      (COND ((> HOPIX 0)                   ;More than one hop to go.
             (MOVE HOPO "RELTO" HOPDIST "STEPS" HOPSTEPS
                   "SHAPE" (CIRC- "T" 3.1416) "FOR" HOPTIM)
             (SETQ HOPIX (- HOPIX 1))
             (RE-SCHEDULE HOPTIM))
            (T (MOVE HOPO "FINISH" SO "SHAPE" (CIRC- "T" 3.1416)
                  "TO" SO"STEPS" (- (STEPS SO) (CURSTEP SO))))))))
```

Hopper uses SCHEDULE to concatenate the hops, "waking up" just as each hop has been completed. Various state variables are initially set up to decrease the computation needed at each wakeup; most are straightforward. HOPDIST, the vector distance between hop start and end points, is computed using ANGLPT, for ANGle PoinT, which is not shown. ANGLPT returns a unit vector (position) whose angle is that between two position arguments. Multiplied by the scalar HOPL, this produces the HOPDIST.

The final example creates spiral motion cycling around an arbitrary curved path, as illustrated in Fig. 6-5. This is done by using a full circular motion creating picture function CIRCL which is rather similar to HOPPER above; it produces circular cycles starting and finishing at the origin and centered at a point on the X-axis. CIRCL's coordinate system is then transformed by an M-DALI ("VAL") daemon to center it at the current position on the input path and rotate it so that its X-axis is tangent to the path at that position. RESPACE, discussed in section 4.2, is then used to translate CIRCL's output into a meaningful position which performs the spiral. The diameter of the circle produced by

CIRCL, which is the diameter of the spiral, is parameterized as SIZ; and the distance travelled forward during each full spiral is parameterized as DIST. The outer picture function, SPIRAL, and CIRCL itself follow.

```
(DEFPIC SPIRAL (SO DIST SIZ "OUTU" SPIRO "AUXO" ROT)
  (OUCH ROT 0)
  (ONC (VAL SO) (ROT)
    (OUCH ROT (ANGLE (PRESTEP SO) (POSTSTEP SO))))
                              ;text following explains these functions
  (SETQ SPIRO
    !(RESPACE
       !(TRANSFORM "CENTER" SO "ROTATION" ROT
                   CIRCL SO DIST SIZ))))

(DEFPIC CIRCL (SO DIST SIZ "OUT" CIRCO "AUX" CYCTIM CYCIX CYCSTEP)
  (OUCH CIRCO (POS 0 0))
  (ONCIF (SEQ SO) (CIRCO)
    (SETQ CYCIX (/ (PATHLENGTH SO) DIST))       ;see following text
    (SETQ CYCTIM (/ (- (FTIME SO) (STIME SO)) CYCIX))
    (SETQ CYCSTEP (/ (STEPS SO) CYCIX))
    (SCHEDULE 0
      (COND ((< CYCIX 1)
             (MOVE CIRCO "RELTO" 0 "SHAPE" (CIR+ "T" SIZ)
                         "FOR" CYCTIM "STEPS" CYCSTEP)
             (SETQ CYCIX (- CYCIX 1))
             (RE-SCHEDULE CYCTIM))
            (T (MOVE CIRCO "RELTO" 0 "SHAPE" (CIR+ "T" SIZ)
                           "FINISH" SO
                           "STEPS" (- (STEPS SO) (CURSTEP SO)))))
      )))

(DEFINE CIR+ (T SIZ)
  (* SIZ (POS (+ (/ SIZ -2) (SIN (* T 6.2831)))
              (COS (* T 6.2831)))))
```

The above uses four functions which will not be explicitly coded: PRESTEP returns the value a sequenced output had on its previous value step; if there was no such step, i.e., the step index is 1, it returns the output's OVAL. POSTSTEP is similar, returning the next step or the OVAL. ANGLE returns the angle in radians a vector between two positions makes with the X-axis; if the positions coincide, it returns 0. These three are used to compute an appropriate rotation. Finally, PATHLENGTH does a simple integration of the distance a two-dimensional value will travel according to its current sequence. This integration does not have to be extremely accurate, since the final MOVE of CIRCL works on an absolute basis guaranteed to reach the correct final value at the precisely right time.

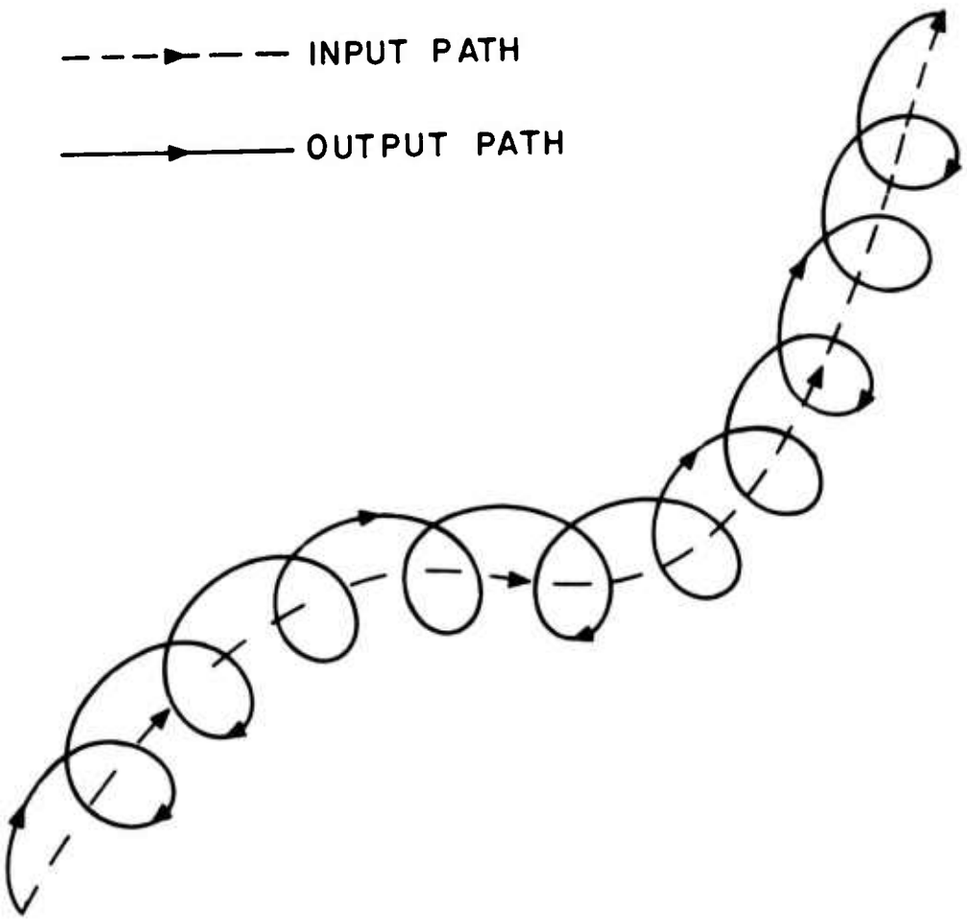


FIGURE 6-5 PATH PRODUCED BY SPIRAL

Chapter 7
Conclusion

7.1 Summary and Conclusions

This document has described DALI, an extension to a programming language's control and environment structures which makes the host language more suitable for controlling changing pictures. The problem was approached by considering not just change, but the propagation of computed changes through the structure of the picture. Change with no temporal structure, that is, change occuring at a monadic instant of time, was considered first and rather exhaustively. This was dealt with by the subset of DALI called M-DALI, whose description include DALI's four primary types of objects -- picture modules, picture functions, outputs, and daemons -- their organization into a containment tree and a data web, the environment structure used, and scheduling rules providing efficient operation and simple inter-"process" cooperation. Deletion, structural change, and functional circularity (relaxation) in M-DALI were then discussed. Change extending over a period of time was then considered in S-DALI, a superset of M-DALI, as temporal sequences of monadic changes.

At its core, DALI consists of the application of two techniques to the problem of describing changing pictures in a manner allowing their effective computation. These two techniques are:

(1) the use of user-written event-driven procedures -- daemons -- as an intrinsic part of the picture itself;

(2) the analysis of the functional relationships among those procedures -- the data web, analyzed by the daemon scheduling rules -- for the purpose of ensuring efficient and harmonious interaction between elements of the picture.

The remainder of M-DALI, i.e., picture modules, local environments, outputs, etc., exists primarily to fully exploit the above two techniques and simplify their utilization as far as possible; and, once the distinction is made between "compute time" and "picture time", S-DALI is a fairly straightforward extension of M-DALI's mechanisms to the problem of describing and organizing (picture-) temporal sequences of pictures.

By its use of the above two core techniques, DALI provides several substantial advantages over methods of picture description currently in use.

The first advantage is flexibility. The majority of existing systems, e.g., instance tree systems (section 1.2), support only a small, fixed set of relationships between picture elements. In comparison, the set of inter-element relationships supported by DALI is essentially infinite, because those relationships are represented in the picture itself by arbitrary user-written procedures. Viewing process systems theoretically also have this capability, but it is effectively cancelled out by problems of synchronization, mentioned in section 1.2, which DALI does not have.

Another advantage is modularity. By exploitation of the first core technique, the embedding of user procedures in the picture, every DALI picture module can support arbitrary computation with local memory; so every module is fully capable of "taking care of itself", and hence the caller of a picture function need not be concerned with how the resultant picture module carries out its endeavors. The same degree of modularity is very difficult to achieve with more limited systems, since to support complex picture change they require the mutual organization

of (1) a picture description, (2) a program to make up the descriptive deficiencies of the picture description, and (3) an often complex data structure to relate the picture description and the program; combining these three in a modular fashion presents major difficulties.

The second core technique, the analysis of the data web by the scheduling rules, is also a critical element in providing modularity. It is primarily the scheduling rules which assure that the separate "processes" -- daemons -- contained in a picture module will not only work as expected, but also will not interact destructively with other "processes" external to the module. That such smooth inter-"process" communication and cooperation is achievable with straightforward, even simpleminded programming containing no reference to semaphores, locks, and the like is as much a virtue of DALI as it is a necessity: since the use of literally hundreds of separate "processes" is implied, DALI would be utterly unusable if this were not the case.

Closely allied to modularity is the fact that DALI provides the graphics programmer with a greatly increased ability to organize and build pictures -- and hence programs -- as a hierarchy of abstractions; this is an ability at least partially available to the general programmer in the form of programmably arbitrary procedures with arbitrary arguments. In comparison, most current systems for dynamic graphics provide only an analog of procedures which can contain no conditional expressions and can take only a fixed set of parameters. Newman's EULER-G system is an exception to this, but it is limited to static graphics.

DALI's increased flexibility and modularity combine to produce extensibility and hardware independence, in the sense that a DALI programmer need not be aware of whether a given picture function was written by another user, was provided by the DALI system in software, or was directly implemented in hardware.

In the light of all the above, it seems somewhat extravagant to

claim that DALI is also a fairly efficient system to run on conventional hardware; but that is the case. Admittedly, there may be spatial overhead, as will be discussed later in this section; but the excess computational overhead is not very large. To establish this, we must ask what DALI does which would not have to be done in any case. For example, if the user really wants 752 logarithmic spirals dancing across the Rocky Mountains, coiling and uncoiling in time with Ravel's Bolero, well, the computation must be done somewhere, and doing it in daemons -- which can be partly or wholly hardware -- is no more intrinsically inefficient than doing it elsewhere.

The primary computational overhead involved in using DALI is found in only two places: (1) each daemon is effectively called as a procedure of one argument, it's owner's local environment, rather than being straight-line code; and (2) daemons must be sorted by priority in the daemon queue. Both of these operations -- procedure calling and sorting -- have a long-established importance outside of the realm of DALI; so currently available techniques can be used to keep this overhead minimal. The time required to establish daemon priorities in a cyclic data web, while it may be substantial, cannot be counted as "overhead"; in this case, DALI is performing an operation which the programmer cannot perform himself, as noted in section 5.1. Furthermore, once the priorities are established, then repetitive calculations using a cyclic data web involves only the two sources of overhead mentioned above.

The efficiency attributed to DALI arises from two sources. The first is the use of the second core technique mentioned above, the analysis of the data web, since it provides the needed scheduling information in the form of easily-used numeric priorities. This can only be done because DALI has available, in the data web, a great deal of information concerning the topology of functional relationships between picture elements. That this information can be provided by the programmer in a manner that is both efficient and not exceptionally arduous is perhaps another advantage of DALI.

DALI's second source of efficiency is its primarily bipartite environment structure, with part of the environment held in a stack and part in heap storage. This is a compromise between fully stack-structured control and environment schemes, epitomized by ALGOL 60, and fully tree-structured schemes, as are, for example, used in CONNIVER [McD1], OREGANO [Ber1], and SIMULA 67 [Dah2]. DALI's scheme retains a much of the efficiency of stacks while providing a good part of the useful flexibility of trees. As was pointed out in section 3.11, however, a modification to DALI's scheme -- involving retention of a daemon's temporary environment across separate executions of that daemon -- may be desireable to provide capabilities closer to those of fully tree-structured schemes. This modification would, however, also entail a decrease in efficiency, oweing to far greater problems of storage allocation. The full facilities of a complete control and environment tree would still not be provided under this modification; however, as was also pointed out in section 3.11, it is not clear that the abilities left out are necessary or even desirable.

At the same time, DALI is not without disadvantages.

First, as mentioned above, there is a certain amount of overhead in storage space which a DALI program incurs over a system specialized to the program's application. To some extent this must be true of any general system. In the case of DALI, the primary source of this overhead is the complexity of outputs, which are nowhere near as small as the author would like.

Second, the overhead involved in changing functional relationships between daemons, i.e., in making structural change to the data web and adding cycles, can be quite high. It is unclear at the present time exactly how much of a disadvantage this is.

Third, the use of deletion as opposed to garbage collection, which is forced as described in section 4.3 and Appendix 3, is a disadvantage, in that it has two demonstrably bad effects which were discussed in

section 4.3: First, deletion nearly doubles the number of system-maintained references (pointers) needed in each element, since nearly every necessary reference requires a corresponding reference in the other direction so that the needed reference can be spliced out if the object referred to is deleted. Second, deletion requires, or at any rate inspires, the use of "implicit dependence deletion" to establish the desired semantics and protect the user from inadvertently accessing storage which has been returned to the "free" pool. About the most charitable thing that can be said of "implicit dependence deletion" is that it sometimes does approximately the right job. Its overhead is high, proportional to the degree of user protection provided, and in general it is decidedly inelegant. However, the author has not quite given up on garbage collection.

7.2 Implementation Issues

Like any other programming language, DALI cannot be considered truly "tested" until it has been applied by users to a body of specific applications. Unfortunately, no usable implementation exists at this writing.

An experimental implementation of a large subset of M-DALI and S-DALI, not including circular data webs or structural change, has been performed using the MIT Project MAC Programming Technology Group's Digital Equipment Corp. PDP-10. The base language used was MUDDLE [Pfi2], a LISP-like language. The purpose of this implementation was to make sure that all the parts of DALI fit together in a reasonable way, and to obtain some idea of the size of the run-time system needed. The latter was gratifyingly small, amounting to approximately 6500 words of rather "loose" PDP-10 program. It included creation of picture functions, picture modules, daemons, and outputs; run-time local

environment accessing; daemon queueing and running; OUCH; and a simplified MOVE. However, this implementation was an interpreter written in an interpreter (MUDDLE), and compares favorably in speed only with continental drift. Recent attempts at compiling the DALI interpreter, however, tend to confirm the comments made in the previous section concerning DALI's efficiency.

At the present time, an implementation is in progress for the MIT Project MAC Control Robotics Group's Digital Equipment Corp. PDP-11/45. It is intended that this implementation be used to create educational films illustrating the operation of complex software such as assemblers, interpreters, and compilers. This implementation will be an extension of ALGOL, hence compiled, and will hopefully provide good feedback on the usability of DALI.

In connection with this implementation, a Master's thesis by C. Terman is exploring the possibility of using the data web as input to modified program optimization techniques. The intent is to eliminate the need for explicitly queueing daemons in structurally static data webs, and in addition allowing the replacement of intermediate outputs with simple value cells which may be dynamically allocated from a stack.

7.3 Directions for Future Research

Various subjects closely gathered 'round about the work presented bear closer investigation. Three examples come to mind immediately: daemon scheduling in cyclic data webs, arbitrary user-definable coordinate transformations, and a true multiple processor implementation of DALI.

Cyclic scheduling is simply a rather difficult problem whose depths have assuredly not been plumbed here. In particular, it would seem that there should be some method of describing cycles in the data web --

e.g., an analog of a "while" loop -- that would both permit more facile control over iteration and permit the DALI system to find out the topology of cycles in a more direct manner. The latter, in particular, would reduce the necessity for the connectivity matrix computations required in the present scheme. The problems here lie in interaction among such loops, particularly in allowing them to intersect one another and ensuring that such multiply-intersecting loops are relaxed as a whole.

Since coordinate transformations and their concatenation are at least conceptually defined in DALI in terms of daemon execution, the possibility exists for allowing the daemons involved to be written by the user. This would open the door to a truly spectacular increase in flexibility; for example, a system might be defined which uses Schwartz-Christoffel transformations [Arf1] to "bend" a half-plane around multiple corners, etc. However, allowing arbitrary user transformations raises severe problems of operating speed, since arbitrary transformations cannot in general be concatenated into a single transformation.

A multiple-processor implementation which took advantage of the parallelism which the scheduling rules allow could produce impressive gains in speed, especially if very local parallel processing were possible -- for example, the equivalent of one (micro-) processor per picture module, possibly shared among daemons and picture modules which necessarily run sequentially relative to one another. Intercommunication is, as usual, a primary problem; another is scheduling for cyclic data webs.

However, there are other issues to be considered: in what direction does DALI lead?

One possible direction is that of making picture elements more intelligent. DALI just barely begins to provide picture elements with the basic necessities for this: the ability to obtain data from "the

outside world", the ability to "reason" about such data via arbitrary procedures, and the ability to affect "the outside world" in turn. It is clear that the abilities provided by DALI are not going to get us very far in this direction; but at the same time, they can be useful. As a graphic example, suppose an object is "commanded" to cross a veritable Maginot Line of obstacles. The task of plotting a path which avoids disaster is clearly out of the realm of DALI-esque techniques, and squarely in areas normally assigned to artificial intelligence; but once the path is plotted, DALI can perhaps help in performing locomotor functions, keeping the moving object a coherent whole, and perhaps guarding against bruised shins and other minor pitfalls.

But how does one think about and describe such spatial reasoning problems? To paraphrase a statement made at the start of Chapter 1: In what way can the spatial relationships in a picture, as opposed to the internal structures used to generate a picture, be organized, grouped and ordered in a way that highlights aspects important to the picture-creating program, supresses details not important to that program, and provides the program with an effective means of computing the visual effects which are so easily conceived -- but not so easily described verbally -- by the programmer?

These are questions more normally ascribed to artificial intelligence. But to the extent that computer graphics solves its basic problem of creating a "super paintbrush" and begins to consider in a more serious, sophisticated manner the problems involved in actually using this medium in new productive ways, they are questions which must be dealt with.

Bibliography

Abbreviations

ACM	Association for Computing Machinery
CACM	Communications of the ACM
FJCC nnnn	Proceedings of the nnnn Fall Joint Computer Conference
IEEE	Institute of Electrical and Electronics Engineers
SJCC nnnn	Proceedings of the nnnn Spring Joint Computer Conference

[Abr1] Abrams, M. D., "Data Structures for Computer Graphics", Proceedings of a Symposium on Data Structures in Programming Languages, ACM SIGPLAN Notices, Vol. 6, No.2, Feb. 1971

[Arf1] Arfken, G., Mathematical Methods for Physicists, Academic Press, Inc., London, 1970

[Bae1] Baecker, R. M., Interactive Computer-Mediated Animation, MAC-TR-61, Project MAC, MIT, Cambridge, Mass., 1969

[Bel1] ---- (film) Incredible Machine (1969), available from Film Library, Bell Telephone Laboratories, Murray Hill, NJ 07974

[Ber1] Berry, D. M., "Introduction to OREGANO", Proceedings of a Symposium on Data Structures in Programming Languages, ACM SIGPLAN Notices (Feb. 1971) Vol. 6 No. 2, pp. 171-190

[Bur1] Burton, R. P., Real-time Measurement of Multiple Three-Dimensional Position [illegible] UC-CSc-72-122, Computer Science Dept., University of Utah, Salt Lake City, Utah 84112, (June 1973)

[Chr1] Christensen, C., Pinson, E. N., "Multi-Function Graphics for a Large Computer System", FJCC 1967, Thompson Books, Washington, D. C., pp. 697 ff.

[Cot1] Cotton, I. W., Greatorex, F. S. Jr., "Data Structures and Techniques for Remote Computer Graphics", FJCC 1968, Thompson Books, Washington, D. C., pp. 533 ff.

[Dah1] Dahl, O.-J., Nygaard, K, "SIMULA - An ALGOL-Based Simulation Language", CACM Vol. 9, No. 9 (Sept. 1966) pp. 671-682.

[Dah2] Dahl, O.-J., Myhrhaug, B., and Nygaard, K., Common Base Language, Publication No. S-22, Norwegian Computing Center, Forskningsveien 1 B, Oslo 3, Norway, Oct. 1970

[Dij1] Dykjstra, E. W., "The Structure of THE Multiprogramming System", CACM Vol. 11, No. 5 (May 1968) pp. 341-346

[E&S1] ----, Line Drawing System Model 1 Reference Manual, U0800-1-1, Nov. 1, 1970; Evans and Sutherland Computer Corp., 3 Research Rd., Salt Lake City, Utah.

[E&S2] ----, Line Drawing System Model 2 Reference Manual, 901002-100, Aug. 1, 1971; Evans and Sutherland Computer Corp., 3 Research Rd., Salt Lake City, Utah.

[Gra1] Gray, J. C., "Compound Data Structures for Computer Aided Design: A Survey", Proceedings of the ACM Twentieth National Conference, Thompson Books, Washington, D. C., (1967), pp. 355 ff.

[Hab1] Habermann, A. N., "Prevention of System Deadlocks", CACM Vol. 12, NO. 7 (July 1969) pp. 373-377, 385

[Hew1] Hewitt, C., Description and Theoretical Analysis of PLANNER, AI-TR-258, Artificial Intelligence Laboratory, MIT, Cambridge, Mass, April 1971.

[Hur1] Hurwitz, A., Citron, J. P., and Yeaton, J.B., "GRAF: Graphic Additions to FORTRAN", SJCC 1967, pp. 553 - 557

[Kno1] Knowlton, K. C., "A Computer Technique for Producing Animated Movies", SJCC 1964, Spartan Books, Baltimore, Md., pp. 67-87.

[Kno2] Knowlton, K. C., (film) A Computer Technique for the Production of Animated Movies, available from Technical Information Library, Bell Laboratories, Murray Hill, NJ 07974

[Kno3] Knowlton, K. C., "EXPLOR - A Generator of Images from Explicit Patterns, Local Operations, and Randomness", Proceedings of the Ninth Annual UAIDE Meeting (1970), Miami Beach, Fla., pp. 543-583

[Kul1] Kulsrud, H. E., "A General Purpose Graphical Language", CACM Vol. 11, No. 4 (April 1968) pp. 247-254

[Lic1] Licklider, J.C.R., "A Picture is Worth a Thousand Words - and It Costs ..." 1969 SJCC, pp. 617 - 621

[Mey1] Meyer, T. H., and Sutherland, I. E., "On the Design of Display Processors", CACM, Vol. 11, No. 6, pp. 410-414.

[McC1] McCarthy, John et al., LISP 1.5 Programmer's Manual, MIT Press, Cambridge, Mass., January 1966

[McD1] McDermott, D., and Sussman, G.J., The Conniver Reference Manual, MIT Artificial Intelligence Laboratory Memo 259, May 1972

[Mon1] Money, J., and Eberhardt, T., Man and Woman, Boy and Girl, The Johns Hopkins University Press, Baltimore and London, 1972

[Moo1] Moon, D. A., MACLISP Reference Manual, Project MAC, MIT, Cambridge, MA, 02139 (April 1974)

[New1] Newman, W. R., "Display Procedures", CACM, Vol. 14, No. 10, October 1971

[New2] Newman, W. R., and Sproull, R. F., Principles of Interactive Computer Graphics, McGraw-Hill Book Co., Inc., New York, N.Y., 1973

[New3] Newman, W. M., and Sproull, R. F., "An Approach to Graphics System Design", Proceedings of the IEEE, Vol. 62, No. 4 (April 1974), pp. 471-483.

[Par1] Parke, F. I., Computer Generated Animation of Faces, UTEC-CSc-72-120, Computer Science Dept., University of Utah, Salt Lake City, Utah 84112, (June 1972)

[Pfi1] Pfister, Gregory F., A Display Processor for Biological Image Processing, MIT Master's Thesis, June 1969.

[Pfi2] Pfister, Gregory F., A MUDDLE Primer, Proj. MAC Programming Technology Division document SYS.11.01, Dec. 15, 1972, Project MAC, MIT, Cambridge, Mass.

[Pfi3] Pfister, Gregory F., "Review of Newman and Sproull's 'Principles of Interactive Computer Graphics'", Information and Control, Vol. 24, No. 3, (March 1974) pp. 299 - 303

[Ram1] Ramamoorthy, C. V., "Analysis of Graphs by Connectivity Considerations" Journal of the ACM Vol. 13 No.2 (April 1966) pp. 211-222.

[Rob1] Roberts, L. G., "Graphical Communication and Control Languages", Second Congress on the Information System Sciences, Spartan Books, New York, 1965

[Rov1] Rovner, P. D., and Feldman, J. A., "The LEAP Language and Data Structure", Proceedings of the 1968 IFIP Congress, Vol. 1, North-Holland, Amsterdam, pp. 579-585

[Ros1] Ross, D. T., "The AED Approach to Generalized Computer-Aided Design", Proceedings of the 1967 ACM National Conference, Thompson Books, Washington, D. C., pp. 367 ff.

[Rul1] Rully, A. D., "A Subroutine Package for FORTRAN", IBM Systems Journal, Vol 7, Nos. 3 and 4, p. 248, 1968

[Smi1] Smith, D. N., "GPL/I - A PL/I Extension for Computer Graphics", AFIPS Proceedings, Vol. 38, Spring 1971 pp. 511-528.

[Spe1] ------, IEEE Spectrum, Vol. 11, No. 2 (Feb. 1974) ("Computer Special" Issue)

[Spr1] Sproull, R.F., and Sutherland, I.E., "A Clipping Divider", FJCC 1968

[Spr2] Sproull, R. F., Proposed Network Graphics Protocol, ARPA Network Information Center NIC No. 19933, NGG No. 5, (Oct. 10, 1973)

[Sut1] Sutherland, I. E., "SKETCHPAD: A Man-Machine Graphical Communication System", AFIPS Proceedings, Vol. 23, Fall 1963, Spartan Books, N.Y., pp. 329-346

[Sut2] Sutherland, I. E., and Hodgman, G. W., "Reentrant Polygon Clipping", CACM Vol. 17, No. 1 (Jan. 1974) pp. 32-42

[SuW1] Sutherland, W. R., Forgie, J. W., and Morello, M. V., "Graphics in Time-Sharing; A Summary of the TX-2 Experience" SJCC 1969

[SuW2] Sutherland, W. R., "The CORAL Language and Data Structure", Computer Display Review, Keydata Corp., Watertown, Mass.

[Tho1] Thomas, E. L., TENEX E&S Display Software, Bolt, Beranek and Newman Inc., 50 Moulton St., Cambridge, Mass.

[vDa1] van Dam, A., and Evans, D., Data Structure Programming System, IFIP Congress, Booklet G, 1968.

[vDa2] van Dam, A., "Data and Storage Structures for Interactive Graphics", Proceedings of a Symposium on Data Structures in Programming Languages, ACM SIGPLAN Notices, Vol. 6, No.2, Feb. 1971

[Wat1] Watson, R.W., et al., "A Display Processor Design", AFIPS Proceedings Vol. 35, pp. 209-272, Fall 1969

[Wil1] Williams, R., "A Survey of Data Structures for Computer Graphics Systems", Computing Surveys, Vol. 3, No. 1 (March 1971), pp. 1 ff.

[Wir1] Wirth, N., and Weber, H., "EULER: A Generalization of ALGOL and Its Formal Definition", Pt. 1 CACM Vol. 9, No. 1 (Jan. 1966) pp. 13-23; Pt. 2 CACM Vol. 9, No. 2 (Feb. 1966) pp. 89-99

Appendix 1:
DALI Functions

Conventions used:

A sample application is provided for each function. Lower-case names are meta-variables, and lower-case names beginning and ending with a hyphen (-) indicate 0 or more objects.

,object
syntactic sugar for (OVAL object).

!object
syntactic sugar for (OUT object).

(AS-NEEDED -body-)
create and return a new daemon; the new daemon has body -body-, specifies all outputs explicitly OUCHed in -body-, and watches with a (VAL -outs-) all outputs whose OVALs (,) are explicitly referenced in -body-.

(CAN-WATCH dem (-outs-))
efficiency measure to avoid excess calls on RE-PRIORITIZE. Makes the priority of dem greater than the largest priority of all specifiers of -outs-.

(CONTIN -body-)
create and return a new daemon, running its body once before returning; the new daemon has body -body-, specifies all outputs explicitly OUCHed in -body-, and watches with a (VAL -outs-) all outputs whose OVALs (,) are explicitly referenced in -body-.

(CONTROLLED seq)
returns the output controlled by sequence seq.

(CURSTEP out-or-seq)
returns the step index of a sequence; out-or-seq can be either a sequence or an output with a current sequence, and in the latter case the output's current sequence is used.

(DEFPIC atm (arg-list) -body-)
"declares" the ATOM atm to be a picture function with argument list (arg-list) and body -body-.

(DELETE obj)
destroy obj and any object which is either part of obj or requires obj for its correct operation.

(DODA n (inits) obj)
create a data web chain of modules; new modules are obtained by evaluating obj, (inits) are the objects to start the chain, and n is an output containing the number of objects to be in the chain.

(FVAL out-or-seq)
return the final value of a sequence; out-or-seq can be either a sequence or an output with a current sequence, and in the latter case the output's current sequence is used.

(IVAL out-or-seq)
return the initial value of a sequence; out-or-seq can be either a sequence or an output with a current sequence, and in the latter case the output's current sequence is used.

(LOOPON cndtn (-specs-) -body-)
create and return a loop daemon; cndtn is its condition, (-specs-) are its specified outputs, and -body- is its body.

(MOVE out -specs-)
create and return a path sequence, applying it to out.

(NAMEDONC name cndtn (-specs-) -body-)
create and return a named-change daemon; name is an atom bound in the temporary environment to a list of the watched outputs whose change was the cause of an execution; (-specs-) are the specified outputs, and -body- is the body.

(NPOUCH out pth stime ftime steps)
create a "naked" path sequence and apply it to output out; pth is the path, stime is the start time, ftime is the finish time, and steps is the number of steps.

(NULLSPEC out)
give out the "null" daemon as a specifier; this daemon never runs, and is equivalent to the driving program in web ancestry.

(ONC cndtn (-specs-) -body-)
create and return a daemon; cndtn is its condition, (-specs-) are its specified outputs, and -body- is its body.

(ONDELETION obj fun)
create and return a deletion p-closure; obj is the object whose deletion will cause it to run, and fun is the function applied to obj.

(ONS cndtn (-specs-) -body-)
create and return a daemon, running its body once before returning; cndtn is its condition, (-specs-) are its specified outputs, and -body- is its body.

(OUCH out newval)
change the value of out to newval and return newval; causes all VAL-condition daemons watching out to run.

(OUT mod n)
return the nth output of mod; n is optional and defaults to 1; an application of OUT with one argument is abbreviated by a prefixed exclamation point (!).

(OUTPUT ival)
create and return a new output with initial value ival.

(OVAL out)
return the value of output out; abbreviated by a prefixed comma (,).

(P-CLOSURE fun mod)
create and return a p-closure of fun with respect to mod's local environment; mod is optional and defaults to the owner of the daemon executing.

(PICTURE (arg-list) -body-)
create and return an unnamed picture function with argument list (arg-list) and body -body-.

(PREOUT integer)
return the integerth output of the previous module on a DODA chain; applicable only in the obj argument of a DODA.

(RE-PRIORITIZE dem)
re-calculate the priority of dem and all web descendants of dem; not a user-callable function, but is rather called by DALI as part of WATCHES and SPECIFIES.

(RELINE pos delt)
a picture function which creates a module drawing a line from pos to pos+delt; both arguments are outputs, and the one output is pos+delt.

(RELP p1 p2)
picture function; create a module whose one output is maintained at the sum of the values of p1 and p2.

(RESPACE out)
picture function; create a module whose one output is a position: the value of out, transformed from its definition space into the space in which the RESPACE is applied.

(SCHEDULE ti -body-)
create a scheduled p-closed action (SPA) and schedule it for the current time plus ti; -body- is the body.

(SEQ -outs-)
not a function, but a daemon condition; causes its daemon to be run whenever a sequence applied to any of the -outs- is initialized.

(SEQUENCE out)
return the current sequence of out.

(SEQUENCE? out)
boolean, returns true if and only if out has a non-null current sequence.

(SOUCH out vals stime ftime)
create a simple sequence and apply it to out; vals is the ordered set of values, stime is the start time, and ftime is the finish time.

(SPECIFIES dem out)
makes dem become the specifier of out; out must have no specifier.

(STEPS out-or-seq)
return the number of steps of a sequence; out-or-seq can be either a sequence or an output with a current sequence, and in the latter case the output's current sequence is used.

(STIME out-or-seq)
return the start time of a sequence; out-or-seq can be either a sequence or an output with a current sequence, and in the latter case the output's current sequence is used.

(SVAL out-or-seq integer)
return the value which will be used by a sequence at its integerth step; out-or-seq can be either a sequence or an output with a current sequence, and in the latter case the output's current sequence is used.

(TRANSFORM -specs- pf -args-)
apply the picture function pf to -args- and return the result; visible output created by pf's module will appear in the area defined by -specs-.

(TVAL out-or-seq ti)
return the value which a sequence will assume at time ti; out-or-seq can be either a sequence or an output with a current sequence, and in the latter case the output's current sequence is used.

(UNSPECIFY out)
cause out to become unspecified.

(UNWATCH dem out)
cause dem to cease watching out.

(VAL -outs-)
not a function, but a daemon condition; causes its daemon to be run whenever any of -outs- is OUCHed.

(WATCHES dem out)
cause dem to watch out for OUCHes.

Appendix 2:
DALI Objects

This appendix contains an alphabetized list of all the objects defined by DALI and their component parts, with a short description of each. For the sake of simplicity, direct two-way pointers are assumed, rather than ring structures.

agenda block a set of actions to be performed at a given picture time

- time the picture time at which the actions are to be performed
- next agenda block pointer to the agenda block with the smallest time greater than this one's
- action set the actions to be performed
  - continuation all sequences of output changes which perform one of their changes at this agenda block's time
  - starters all sequences of output changes which are to be initialized at this agenda block's time
  - scheduled p-closed actions arbitrary user-chosen code sequences to be executed at this agenda block's time

daemon a parameterless procedure to be executed in response to some event

- owner the picture module whose local environment this daemon uses; also, the picture module owning the daemon which created this daemon
- type the type of daemon this is, i.e., plain, loop, or named-change
- condition the event whose occurrence causes this daemon to run
  - condition type the type of output change this daemon responds to; this can be an instantaneous value change or the application of a sequence of such value changes
  - watched outputs changes to these outputs cause this demon to run

body the body of code to be executed when this daemon is run

specified outputs the outputs whose values this daemon changes

cyclic priority used to determine when this daemon runs relative to other daemons; see section 5.8

acyclic priority used to determine when this daemon runs relative to other daemons; see section 5.8

deletion triad defined elsewhere in this appendix

changed-outputs list this element exists only if the daemon is a named-change daemon: it is a list of those watched outputs whose values have changed since the daemon was last run

name-identifier this element exists only if system is interpretive and this daemon is a named-change daemon: it is the identifier to be bound to the watched-outputs list just before the daemon is run

deletion triad this is not a separate object, but rather three elements present in all deletable objects

mark bit used to prevent infinite recursion when "recursively" deleting in cyclic data webs

deletion p-closures actions to be performed just before the object containing them is finally destroyed, i.e., interred

dependent modules modules containing a reference to this object in their local environments; unless these modules remove such references (via deletion p-closures) they will be deleted when this module is deleted

deletion p-closure an action to be performed just before an object is deleted

corpse the object whose deletion triggers performance of this action

p-closure the action performed consists of applying this p-closure to the corpse

deletion triad deletion p-closured can be deleted, too

module an organizational unit providing local storage and hierarchical structure

father the picture module containing this one, i.e., the owner of the daemon which created this picture module

sons the picture modules created by daemons owned by this module

local environment local storage for the daemons owned by this module

output identifiers bindings for those identifiers whose values can be accessed from outside this module by use of OUT

other identifiers all other bindings in this local environment

owned objects all the daemons, outputs, deletion p-closures, path sequences, scheduled p-closed actions, and simple sequences owned by this module

deletion triad defined elsewhere in this appendix

output a place to put data such that daemons can detect changes in that data

owner the module owning this output

value the data in this output

specifier the daemon allowed to change this output

current sequence the (S-DALI) sequence of value changes which currently controls this output's value, if such a sequence exists

S-watchers the daemons watching for the application of entire sequences of changes to this output

M-watchers the daemons watching for instantaneous value changes to this output

deletion triad defined elsewhere in this appendix

p-closure DALI equivalent of a functional closure

owner the module which owns this p-closure; also the module whose local environment will be used when the p-closure is applied to arguments

function the function of which this is a closure

deletion triad defined elsewhere in this appendix

path sequence a sequence of changes to be applied to an output, defined as the composition of a transformed shape path and a time path

owner the picture module owning this path sequence

changee the output controlled by this path sequence

path the composite function defining the sequence of values to be assumed by the changee

time path function specifying the rate of travel along the shape path

function the function to be applied to a real number between 0 and 1 and to other constant arguments

to obtain another number between 0 and 1 to which the shape path function is applied

extra data the constant arguments mentioned above

shape path the function defining the spatial path of the sequence

function the function to be applied to the output of the time path and other constant arguments to obtain a spatial position

extra data the constant arguments mentioned above

translation the translation to be applied to the output of the shape path function to obtain the desired position

rotation the rotation to be applied to the output of the shape path function to obtain the desired position

step count the number of incremental value steps this path will produce

step index an integer indicating which of the incremental value steps this path will next produce

step duration the picture time interval to elapse between successive incremental value steps

agenda block the agenda block containing this path sequence, either as a starter or as a continuation

deletion triad defined elsewhere in this appendix

picture function a function which, when applied to arguments, creates and returns a new picture module

argument list defines the number of arguments the picture function takes and also defines the local environment of the resultant picture module

body the body of code to be executed when this picture function is applied

scheduled p-closed action an arbitrary user-defined action to be performed at a specific picture time

owner the module which owns this scheduled p-closed action; also the module whose local environment will be used during the performance of the action

body the body of code evaluated (executed) to perform the action

agenda block the agenda block containing this action in its action set

deletion triad defined elsewhere in this appendix

simple sequence a sequence of values to be taken on by an output, defined by simply listing those values

owner the picture module owning this simple sequence

changee the output whose values are specified by this simple sequence

value set the ordered list of values which this sequences causes the changee to assume

step index an integer indicating which of the set of values this simple sequence will next produce

step duration the picture time interval to elapse between successive value changes

agenda block the agenda block containing this path sequence, either as a starter or as a continuation

deletion triad defined elsewhere in this appendix

Appendix 3:
Garbage Collection in DALI

The purpose of this appendix is to give further details of the garbage collection (retentive storage management) scheme assumed in section 4.3, where it is stated that DALI is incompatible with garbage collection.

The principle property we wish a retentive storage management scheme to have is:

> If a reference to an object is not explicitly retained by a program (daemon), that object is reclaimable as garbage.

Only the reclamation of storage is to be taken over by the system; the user must have full control over the generated picture, e.g., controlling the visibility of the picture by varying the intensity of not-yet-garbage collected objects.

As an illustration of what is meant by an explicitly retained reference, note that this picture function for a triangle drawing picture module will not work with garbage collection:

```
(DEFPIC TRIDELETE (P1 P2 P3)
        (LINE ,P1 ,P2) (LINE ,P2 ,P3) (LINE ,P3 ,P1))
```

Since no references are explicitly retained to the LINEs, they will be garbage-collected at the first opportunity. Under garbage collection, such a module must be written as

```
(DEFPIC TRIGARBCOL (P1 P2 P3 "AUX" EDGES)
        (SETQ EDGES (LIST
         (LINE ,P1 ,P2) (LINE ,P2 ,P3) (LINE ,P3 ,P1) )))
```

thereby retaining references to the LINEs in a list bound to EDGES. Such a retention will keep the LINEs from being garbage-collected unless the entire TRIGARBCOL module is garbage-collected, which is fair enough.

The LINEs of the above example bring to the fore an important aspect of garbage collection in DALI which does not appear in most garbage-collected systems: the need to keep a separate memory space congruent with garbage-collected space. Specifically, garbage collection of a LINE module must somehow cause the LINE's entry in the hardware display file to be destroyed.

If the hardware display file occupies the same logical memory as the rest of DALI storage, it is conceivable that it could be garbage collected in conjunction with garbage collection of the "normal" storage space. But the complexities of doing so are quite large, and in any case this cannot readily be done if the physical display hardware storage is remote.

To solve this problem, garbage-collection p-closures (GCPCs) are used. These are analogous to the deletion p-closures (DPCs) mentioned in section 4.3. A GCPC has, like a DPC, a corpse which refers to it, an owner, and a body. The body is run just before the storage of the corpse is to be reclaimed, and can thereby cause the external storage associated with the corpse to be reclaimed.

The correct operation of GCPCs requires several things:

(1) All GCPCs for a given garbage collection must be run before any storage is actually reclaimed, since reclamation generally entails modifying the contents of reclaimed storage in order to thread it onto a "free list", and GCPCs will often need those contents -- e.g., the environment of a reclaimable LINE module. This means that there must be a separate phase of garbage collection for GCPC operation, placed between the traditional mark and sweep phases which, respectively, identify non-reclaimable storage and reclaim everything else.

(2) Obviously, a GCPC cannot utilize heap ("free") storage; it must, however, be allowed to use some storage -- e.g., FIFO stack storage also used by the garbage collector itself.

(3) The garbage collector must know when it is reclaiming the

storage of an object which has a GCPC. This is not a trivial requirement, since garbage collectors usually do not need to examine reclaimable storage; after all, it's garbage. The required examination can be a part of the extra phase mentioned in item (1) above.

User-definable GCPCs are not a necessity. Since there will generally be a small, fixed number of objects like LINE (e.g., DOT, CURVE, SURFACE, etc.) which use non-garbage-collected storage, they could be specially constructed and their GCPC's effectively built into the garbage collector proper. Here, as in deletion, the more flexible and device-independent solution has been chosen. However, since GCPCs are actually needed only in special cases, it is reasonable to restrict the types of objects which can have GCPCs to picture modules.

The strict lack of a need for user-definable GCPCs under garbage collection is in contrast with the analogous role of DPCs in "delete" DALI: deletion p-closures are strictly necessary in order to obtain modularity, since without garbage collection of user-created structures, something must be done to reclaim such storage and the system cannot do it.

With the above problem solved, the question of modifications to the previously presented DALI structure must be addressed. This modification consists principally of removing references which are only needed for deletion, while retaining all references needed for correct operation. The modification will be presented by walking through a simple example: the relative position module RELP introduced in section 3.1. The RELP picture function is repeated here for convenience:

```
(DEFPIC RELP (PT DISP "OUT" SUM)
        (ONS (VAL PT DISP) (SUM)
             (OUCH SUM (+ ,PT ,DISP)) ))
```

The minimal structure now needed is illustrated in Fig. Ap-1.1. The reason for each reference is:

(1) The watched outputs must refer to the daemon, so that the daemon can be queued to run on output changes.

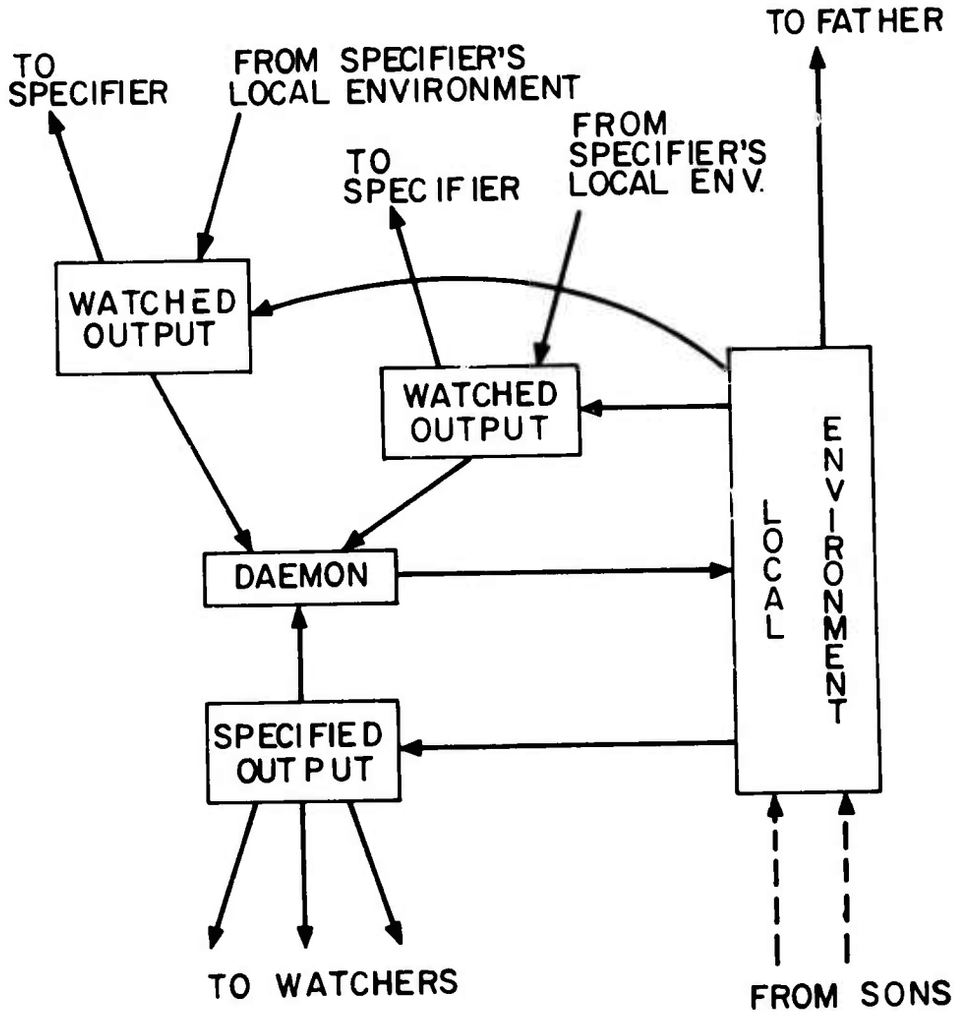


FIGURE AP3-1 REFERENTIAL STRUCTURE OF A RELP MODULE

(2) The daemon must refer to the module's local environment to define its running environment.

(3) The local environment must refer to the watched outputs so that the daemon can access their values.

(4) The daemon must refer to its specified output, at least through the local environment as shown, so that it can change the output's value.

(5) The specified output must refer back to its specifying daemon to establish the data web.

(6) The module must refer to its father, so that "EXTERNAL" inputs of created sons can be looked up. (In the specific case of RELP, this is unnecessary since it creates no sons.)

(7) The module must refer to its GCPC, so that reclamation of non-garbage-collected storage can be done. (In the specific case of RELP, this is again unnecessary.)

The necessary references cataloged above contain a circular path: watched output to daemon to environment to the same watched output. If this were followed by the garbage collector, it would keep a daemon (and its owner, its specified output, its owner's father, etc.) from being reclaimed until all its watched outputs were reclaimed.

This is untenable. Using this circular path, a path can be always traced from the driving program -- never garbage-collected -- to every output and daemon ever created; thus nothing is ever reclaimed. Furthermore, the user cannot break such links himself without detailed knowledge of how other modules are constructed, implying a substantial loss of modularity. The system cannot help him without maintaining the equivalent of all the references deletion requires, thereby compounding the difficulty by introducing further circularities. So this circularity must not be followed by the garbage collector. Where should it be broken?

The environment should not be reclaimed unless the daemon is, or the daemon cannot run.

The environment must refer to the output itself, not just to its value, since the daemon must be able to pass the output to others via, for example, applying picture functions. If the output is reclaimed without the daemon going too, this cannot be done.

That leaves the link from output to daemon. Happily, the daemon is unnecessary for the watched output's continued operation, so we break it there: if no references but those of outputs' dependent daemons exist for a daemon, it should be reclaimed.

So, the garbage collector must not follow the references in an output's list of dependent daemons; the list itself, as distinct from its members, should, however, be marked if the output is marked. Further, the list must be updated (and some elements reclaimed) if daemons in it are reclaimed. This can be cone correctly by means of an implicit GCPC on every daemon, provided we add to daemons references to their watched outputs so that the appropriate sets can be found. These latter back references are followed in the mark phase, as are all references except those in an output's list of dependent daemons.

This sets the stage for the argument of section 4.3: Since daemons are not removed from their watched outputs' dependent daemon sets until the garbage collector runs -- which is hopefully an infrequent occurrence -- they will will be queued and run after the user has done everything feasible to destroy them. Since this can go on for a long period, the daemons can encounter (and produce) anomalous conditions and possibly err catastrophically.

Further problems could be raised, for example: (1) Should not a daemon be reclaimed if the only reference to him is an entry on the daemon queue? (2) What about the S-DALI agenda?

These latter problems are at least potentially solvable. However, given the unsolved problem of making sure daemons are not run when they have been cast off, further discussion of other problems is useless.

Biographical Note

Gregory Francis Pfister was born on November 29, 1945, in Detroit, Michigan. He grew up in Manhasset, Long Island, New York, and attended St. Mary's Grammar School and St. Mary's High School in Manhasset. In 1963, he received a Grumman Aerospace Corp. four-year full-tuition college scholarship (competitive). In September, 1963, he entered Massachusetts Institute of Technology, obtaining the degree of Bachelor of Science in Electrical Engineering in June, 1967. He was co-recipient of the first annual MIT Levin Award (best Junior laboratory project) in 1966.

In September, 1967, he entered Graduate School at Massachusetts Institute of Technology, obtaining the degree of Master of Science in Electrical Engineering in June, 1969; at that time he became an Instructor in the MIT Electrical Engineering Department. Between then and 1974 he taught courses in logic design and programming linguistics at MIT, and worked (summers) at the MIT Lincoln Laboratories Digital Computers Group (TX-2) in Concord, Mass., and at Evans and Sutherland Computer Corp. in Salt Lake City, Utah.

In September, 1974, he received the degree of Doctor of Philosophy in Computer Science at MIT.

At present (1974-1975) he is employed at the IBM Advanced Systems Development Division in Mohansic, N.Y. while on a one-year leave of absence from the University of California at Berkeley, which he will join in the fall of 1975 as an Assistant Professor.

He has been elected to membership in Eta Kappa Nu, Tau Beta Pi, and Sigma Xi.

He has also published a review of Newman and Sproull's Principles of Interactive Computer Graphics [Pfi3], and A MUDDLE Primer [Pfi2].